Performance Analysis of Rainbow on ARM Cortex-M4

Evaluation of a Post-Quantum Multivariate Signature Scheme on a Resource-Constrained Device

Wissenschaftliche Arbeit zur Erlangung des Grades
B.Sc.
an der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München.

Betreut von	Prof. Dr.-Ing. Georg Sigl
	Thomas Schamberger, M.Sc.
	Lehrstuhl für Sicherheit in der Informationstechnik
Eingereicht von	Joan Moya Riera
	Josep Trueta 10
	08970 Sant Joan Despi, Barcelona
	+34 680 31 85 87
Eingereicht am	München, den 30.7.2019

The risk posed by a fully operational quantum computer has anticipated a revolution in the way to approach the level of security provided by a cryptographic algorithm. Public key-based solutions such as RSA or ECC will be easily broken once we enter the post-quantum era. Multivariate quadratic cryptosystems are a promising candidate for the need of quantum resistant digital signature schemes. In order to estimate if these approach will someday be able to replace current standards, it is necessary to determine how efficiently can they operate on diverse platforms and at which level of security can they do it. This aspects are particularly relevant for reduced size devices with restricted energy, memory or computational power.
In this work, a theoretical description of the so-called Rainbow multivariate signature algorithm is given, which is later implemented on a memory-constrained environment. An optimization approach is proposed in order to improve the efficiency of the scheme, in terms of message signature and verification speed. A performance comparison is also presented between various state-of-the-art post-quantum signature cryptosystems and the optimized instances of Rainbow, in order to study its characteristics from a wider perspective.

Contents

1 Introduction

Cryptography is essential for providing security over the exchange of data between two or more communicating parties. In modern society, we have become frequent users of cryptographic techniques, in tasks such as online shopping, mail accessing, website visiting or file downloading. Not only for private usage, but also institutions like banks, governments or the military protect their data by making use of cryptographic solutions. Furthermore, given the growth in the number of services offered on the cloud, it is expected that the number of secure algorithms keeps increasing in the future.

Most asymmetric key cryptosystems currently used in practice are based on large integer factorization and discrete logarithm calculation. Cryptographic approaches that base their security on these mathematical operations are considered to be unbreakable using current computing systems. However, since the invention of Peter Shor's algorithm [Sho99] in 1997 for efficiently solving both problems, the security community has been forced to develop new cryptographic solutions to anticipate a post-quantum cryptography standarization process, before the appearance of a sufficiently large quantum computer.

Post-quantum candidate algorithms are based in mathematical problems that are supposedly resilient against classical, as well as post-quantum cryptanalysis. These quantum immune techniques also need to be feasible for all kind of systems, including small embedded micro controllers. Due to the limited amount of resources on such platforms, the implementation of these solutions to provide post-quantum security on such devices does sometimes lack of feasibility.
Current research related to post-quantum cryptography has been mainly focused on the security provided by the different post-quantum cryptographic solutions. The feasibility of these approaches can be validated by studying the computational complexity of the algorithms and testing their efficiency experimentally.

This work is based on a practical study of the performance concerning the Rainbow multivariate signature cryptosystem on an ARM Cortex-M4 microcontroller. The efficiency is characterized in terms of execution speed and the memory consumption of the scheme, as well as the reachable security levels in the constrained device where the analysis is performed.

The contribution of this thesis is concluded as follows:

1. We present an implementation of the Rainbow multivariate signature scheme, which provides post-quantum security on a suitable device for the Internet of Things.

2. We evaluate the feasibility of the different parameter sets available to provide a specific level of security on our platform.

3. We analyze the performance in terms of execution time and memory consumption of the scheme's different functionalities, considering different compiler optimizations.

4. We propose an optimized solution with increased efficiency on our target device.

5. Based on the observations, we compare the behaviour on the same computing architecture of various Rainbow instances with different state-of-the-art quantum secure digital signature cryptosystems.

Organization. The research and experimental work that has been carried out in the thesis is organized as follows. In Section 2, we introduce post-quantum cryptography and describe the characteristic features of the multivariate approach. In Section 3, we give an overview on finite field algebra, focusing on two aspects that are later exploited during the optimization procedure. In Section 4, we present the construction methodology and the different algorithms provided by the Rainbow signature scheme. Section 5 gathers all the aspects related with the experimental part of the project. The obtained benchmarks are evaluated and discussed in Section 6. Finally, we conclude with some remarks in Section 7, based on the observed results.

2 Post-Quantum Cryptography

2.1 Introduction to PQ-Cryptography

Quantum computers are high-speed parallel computing systems, whose properties and operation principles are based on the behaviour of subatomic particles ruled by quantum mechanics. This systems use qubits instead of bits, which can exist in any superposition of the 0 and 1 states in a conventional device. This principle grants the ability to perform calculations simultaneously, solving problems that require an enormous amount of computing power in current available computers. With operational tasks being drastically accelerated, quantum computers offer promising solutions to complex problems in different fields such as generating sophisticated market models, creating new medicines or exponentially increasing database search speed, among others.

Current quantum computers are far from their capabilities in terms of efficiency, as they operate at temperatures close to the absolute zero. Nevertheless, there are tasks that quantum computers can fulfill in a more efficient way than classical computers do. One of the prominent applications for quantum computers is decoding traditional cryptographic protocols in polynomial time. In particular, there exist two main algorithms that executed in a quantum computing system would be able to break the security of most asymmetric cryptography approaches employed nowadays.

Shor's Algorithm was invented by Peter Shor in 1994. It is a method [Sho99] capable of efficiently solving some of the problems that back up the security of the most popular and currently used public key cryptosystems, based on integer factorization and (elliptic curve) discrete logarithm. All these mathematical trapdoor functions are considered to be hard to solve by classical computing systems, even though a quantum computer with sufficiently powerful resources could break them in polynomial time.

Grover's algorithm consists in another quantum approach [Gro96] that can invert functions in $\mathcal{O}(\sqrt{n})$ time. This algorithm reduces the security of a symmetric key cryptosystem by a root factor when being executed in a quantum computer. However, it is not considered a major threat as doubling the level of security of the scheme can mitigate these attacks.

The future construction of a quantum computer capable of executing these algorithms is quite unknown as only speculations have been made about the moment when such systems

could be brought to light. Nevertheless, in 2006, NIST proposed a competition[1] in order to enhance the development of new candidates to replace RSA and ECC and implement post-quantum cryptography as the standard security model.

Post-quantum cryptography is the study of cryptosystems that can be implemented in a classical computing systems while being able to protect against quantum computer attacks. Post-quantum schemes provide confidentiality, integrity, authenticity, and non-repudiation, as well as traditional cryptosytems do. There exists 6 different types of approaches [LSY+16], which are briefly described below.

- **Lattice-based**
  Lattices are considered the most flexible approach among the others because they provide strong security reductions and are also capable of performing key exchange as well as generating digital signatures. The trapdoor of this scheme is based on solving a system of linear equations that contains an error variable which gets bigger at each step of the Gaussian Elimination algorithm in order to solve the system, until the point where any useful information about the secret key gets hidden. This is called the Learning With Errors problem (LWE), based on finding the shortest vector in a lattice, which is considered to be NP-hard.

- **Multivariate**
  Such schemes base their security both on the NP-hardness of solving systems of multivariate equations over finite fields and the Extended Isomorphism of Polynomials. The size and the speed at which the signatures are generated by this approach, place multivariate cryptosystems as a potential candidate for post-quantum authentication schemes. However, the length of the public and secret keys generated notably reduce the feasibility of this approach, specially on low-resource embedded platforms.

- **Elliptic curves / Isogenies**
  Elliptic curve cryptography relies on the hardness that involves finding the discrete logarithm of a random elliptic curve element with respect to a publicly known base point. The size of the elliptic curve determines how much difficult the problem turns out to be. The main advantage is the small size of the keys generated, respect to other cryptosystems that employ finite fields (discrete mathematics), providing the same level of security.
  Elliptic curve cryptosystems can be broken by quantum computers by performing discrete logarithm operations. In order to provide a secure approach, a supersingular isogeny-based scheme is developed. In this case, the security is based on supersingular elliptic curves in order to create a modified Diffie-Hellman key exchange that can be used as a replacement for the Diffie-Hellman and elliptic curve Diffie–Hellman methods used nowadays. This scheme provides extremely small key sizes even though it is the slowest algorithm for key generation and shared secret acquisition.

[1] `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography`

- **Hash-based**
  Hash-based signature techniques use the input to a hash function as the secret key and the output as the public key. The generated digital signatures by this approach have been studied ever since as an interesting alternative to other authentication approaches, like RSA and DSA, that are based on number theory. For any hash-based public key, there exists a limit that constrains the number of signatures that can be generated using a single set of private keys. This fact reduced the interest on this cryptographic technique until post-quantum cryptography was introduced, given the resistance that they oppose against quantum computer-based attacks. Hash signatures are not memory efficient. However, approach is considered to be fast and the security relies on the underlying hash function

- **Code-based**
  During the transmission of binary messages, there is a certain probability that errors occur by getting the bit flips. Error-correction codes provide the possibility to detect and revert a certain number of bit flips at the expense of message compactness. The most prominent type is called linear codes, represented by matrices, whose size depends on the length of the original message and the length of the encoded message. It is computationally expensive to decode a message without information about the underlying linear code. This problem backs up the security of the so-called McEliece public key cryptosystem. The secret key of this cryptosystem is a random code from a class called Goppa codes. The public key consists of an invertible matrix with binary entries. Like lattices, code-based cryptographic schemes suffer because of the size of the matrices that are used to represent the public and secret keys. Altough, various modifications have been done to the original scheme to improve this aspect.

2.2 Multivariate Public Key Cryptosystems (MPKC's)

Multivariate cryptography is a post-quantum asymmetric key approach based on multivariate polynomials over a finite field. The security of this cryptographic technique relies on both the $\mathcal{MQ}$-problem and the Extended Isomorphism of Polynomials (EIP). The former has been proven to be NP-complete [GJ90] and doubly exponential, even in the simplest case, using a finite field composed by two elements. The latter is related to the construction principle of multivariate schemes.
Unfortunately, there is no security proof regarding the problem regarding the followed procedure to build $\mathcal{MQ}$ cryptosystems. For this reason, most MPKC's have been broken except the Unbalanced Oil and Vinegar signature scheme [KPG99] and its variants Rainbow [DS05] and enTTS [YC05].

2.2.1 Construction of MPKC's

The standard construction, also called **bipolar**, of a multivariate cryptosystem, requires an easily invertible polynomial system of m multivariate quadratic equations in n variables (central map) $\mathcal{Q} : \mathbb{F}_q^n \to \mathbb{F}_q^m$. Right after, two affine maps $\mathcal{S} : \mathbb{F}_q^m \to \mathbb{F}_q^m$ and $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ are chosen. All the scalar parameters and variables that describe the polynomials are randomly chosen within a finite field $\mathbb{F}$ of size q.

The **public key** is computed as the composition between the central map and the affine transformations $\mathcal{P} = \mathcal{S} \circ \mathcal{Q} \circ \mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^m$. As a result, the public key becomes a hardly invertible set of polynomials without any trace of the center map in it, whose structure has been completely hidden among the equations after obfuscating the multivariate equations system $\mathcal{Q}$ with the affine transformations $\mathcal{S}$ and $\mathcal{T}$.
The central map can be modeled as a matrix of multivariate quadratic polynomial equations represented by the following the structure:

$$\mathcal{P} = \begin{bmatrix} p^{(1)}(x_1, x_2, \cdots, x_n) \\ p^{(2)}(x_1, x_2, \cdots, x_n) \\ \vdots \\ p^{(m)}(x_1, x_2, \cdots, x_n) \end{bmatrix}$$

$$p^{(m)}(x_1, ...x_n) := \sum_{j=1}^{n} \sum_{i=1}^{n} p_{ij}^{(m)} x_i x_j + \sum_{i=1}^{n} p_i^{(m)} x_i + p_0^{(m)} = x^T \mathfrak{P}^{(m)} x$$

where $\mathfrak{P}^m$ is the matrix of size $n \times n$ that describes the quadratic form of the polynomials in the public map. The linear and constant terms could be neglected from the public key equation system as they are never mixed with the quadratic terms and therefore, don't contribute to the security of the scheme. This procedure reduces the memory required to store the entire keys of the cryptosystem.

$$x^T \mathfrak{P}^{(m)} x := \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} p_{1,1}^m & p_{1,2}^{(m)} & \cdots & p_{1,n}^{(m)} \\ & \vdots & \ddots & \\ p_{n,1}^{(m)} & p_{n,2}^{(m)} & \cdots & \gamma_{n,n}^{(m)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

The equation system that describes the public key must be hard to invert without the knowledge of a secret. Knowing $\mathcal{S}, \mathcal{Q}, \mathcal{T}$ and their structure, allows the possibility of inverting the obfuscated polynomial system. Thus, the **private key** is formed by the coefficients that describe the equations of the central map and the affine transformations.

Bipolar Constructions

Most multivariate cryptosystems have been built following the bipolar methodology described in 2.2.1. Depending on the structure given to the central map $\mathcal{Q}$, there exists three different alternatives to the standard construction can be differentiated.

- **Single Field**
  The computations are made between elements of a relatively small Galois field. The number of equations m (domain) is chosen smaller than the number of variables n (codomain) and the central map describes a surjective function. This fact restricts this type of construction to signature schemes. By creating such surjection, all the elements from the domain are mapped to, at least, one element in the codomain.
  In this case, the trapdoor is achieved by following a special algebraic pattern to assemble the private polynomials of the central map, instead of using field extensions. In Figure 2.1 it is described the methodology followed for computing the encryption and signature functions in a small field approach.

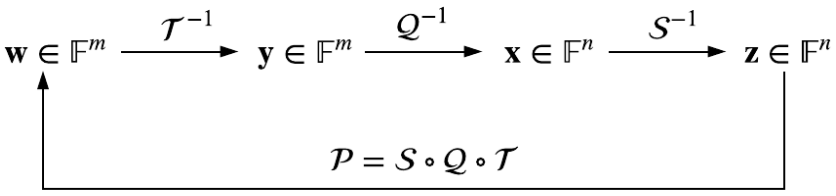

Figure 2.1: Encryption and signature authentication in the lower path and decryption and signature generation procedures in the upper path, following the bipolar construction.

- **Big Field**
  In this case, the number of equations m equals the number of variables n. Thus, the central map is **bijective**, which implies that each element from the domain is mapped to a single element of the codomain. Big Field constructions can be used to design both encryption and signature schemes. It is defined a degree n extension field $\mathbb{E}$ of $\mathbb{F}$ by $\mathbb{E} = \mathbb{F}[X]/g(X)$, being $g(X)$ an irreducible polynomial from $\mathbb{F}[X]$, and the transformation $\Phi : \mathbb{F}^n \to \mathbb{E}$. Once at this point, the central map is defined as $\mathcal{Q} = \Phi^{-1} \circ \widehat{\mathcal{Q}} \circ \Phi$ where $\widehat{\mathcal{Q}} : \mathbb{E} \to \mathbb{E}$. Finally, the public key is computed as the composition between the affine transformations $\mathcal{S} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ and $\mathcal{T} : \mathbb{F}_q^n \to \mathbb{F}_q^n$ and the modified central map, which is expressed as $\mathcal{P} = \mathcal{S} \circ \mathcal{Q} \circ \mathcal{T} = \mathcal{S} \circ \Phi^{-1} \circ \widehat{\mathcal{Q}} \circ \Phi \circ \mathcal{T}$.

- **Middle Field**
  The difference between Big Field and Middle Field constructions is that the latter uses an extension field of degree $\frac{n}{k}$, being k an integer, instead of n, as in Big Field con-

structions. The number of equations m and the number of variables n is equal which implies that this construction can also be used for designing encryption or signature schemes.

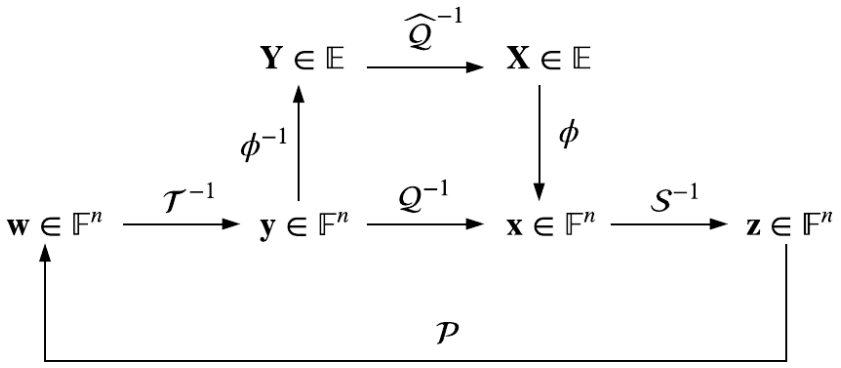

Figure 2.2: Modification of the Single field methodology for Extension field encryption and authentication.

In Big and Medium Field schemes, vector spaces and **hidden field** structures, also called extensions of $\mathbb{F}_q$, are employed to generate the trapdoor. The parameters and variables of the invertible quadratic central map are searched directly in the extension field [WYHL06]. The illustration in Figure 2.2 shows the modification that is made over the single Field construction in order to build a scheme based on field extensions.
In the following lines, it is described the encryption methodology followed in Big and Middle Field constructions, and the message signature procedure carried out in Single Field construction in $\mathcal{MQ}$ cryptosystems.

Encryption: Big and Middle Field ($m \geq n$)

The standard **encryption** procedure in MPKC's, consists in computing the public key polynomial system with the plaintext **z** as the input, in order to obtain a ciphertext **w**. Otherwise, for **decrypting** a given ciphertext **w**, the decrypting person has to compute $y = \mathcal{T}^{-1}(w) \rightarrow x = \mathcal{Q}^{-1}(y) \rightarrow z = \mathcal{S}^{-1}(x)$ recursively in order to get the associated plaintext. Examples of multivariate cryptosystems built for encryption are TTM-related schemes [TC01], ZHFE [PBD14] or PMI+ [Che06].
With **encryption and decryption** schemes, the number of equations **m** is chosen bigger than the number of variables **n**, in order to have an almost **injective** transformation. By doing this, the probability of mapping two different plaintexts to the same ciphertext is reduced, as the public map becomes a close approximation to a one-to-one transformation. In other words, it will rarely happen to compute the preimage and obtain a wrong plaintext.

Signature: Single Field ($m < n$)

In order to **sign** a message, the signer hashes the associated plaintext, obtaining **w** and computes $y = \mathcal{T}^{-1}(w) \rightarrow x = \mathcal{Q}^{-1}(y) \rightarrow z = \mathcal{S}^{-1}(x)$ recursively, to acquire the corresponding signature **z**. To check the **authenticity** of a signature **z** for a given message **d**, the public key is used to map the signature as it is done for the encrypting procedure $w = \mathcal{P}(z)$. Right after, it is verified that the hash of the message to verify equals the decrypted signature $\mathcal{H}(d) = w$. The most commonly known multivariate cryptosystems built for authentication are Sflash [PCG01], HFEv [CDF02] or UOV-based schemes like Rainbow [DS05] and TTS [YC05].
Unlike the encryption procedure, in **signature and verification schemes**, the number of variables **n** in the domain is chosen bigger than the number of equations **m** in the codomain. This practice ensures that every hashed message will be mapped to a signature, even though more than one different digests can share the same image in the codomain. In this case, the public key describes a **surjective** transformation.

2.2.2 Security Principles (Underlying Problems)

Without the knowledge of the private key, an attacker that wants to decrypt a ciphertext that has been encrypted by a multivariate cryptosystem has basically two alternative paths. On the one hand, he can try to solve the system of equations given by the public key ($\mathcal{MQ}$ problem), while on the other hand, he can make an attempt to find the underlying structure of the secret key to break the algorithm. In this case, the attacker dives into a different problem caused by the bipolar construction approach (Extended Isomorphism of Polynomials).

- **MQ Problem**
  Finding $(x_1^{'}, \cdots, x_n^{'})$ such that $p^{(1)}(x_1^{'}, \cdots, x_n^{'}) = \cdots = p^{(m)}(x_1^{'}, \cdots, x_n^{'}) = 0$ is satisfied, where $\mathcal{P} = (p^{(1)}, \cdots, p^{(m)})$ in the variables $(x_1, \cdots, x_n)$, is known as the multivariate polynomial (MP) problem. This problem has been proven to be **NP-hard** even for the simplest case [YDH+13], using a finite field of size 2. Employing quantum computing, the best known attack solves this problem in exponential time, while integer factorization is teared apart in subexponential time.
  Almost all multivariate cryptosystems use second degree polynomials for two main reasons. One is related to the efficiency, as using higher order terms makes the number of coefficients increase rapidly producing unnoticeable efficiency gain. The other concerns the principle that any set of high degree polynomials can be rewritten as a set of quadratic equations. When second degree polynomials are used, the MP problem changes its name to $\mathcal{MQ}$ problem. Attacks made against this principle are called after Direct attacks.

- **EIP Problem**
  Because of the construction of the schemes, the security of MPKC's is not only backed up by the $\mathcal{MQ}$ problem but also by the **Isomorphism of Polynomials** problem. There exists three different versions of IP's. When the central map is unknown, as in most multivariate cryptosystems, the problem is based on the third version of this problem, known as **Extended IP**.
  Knowing the structure of the public key $\mathcal{P}$, an attacker will find nearly impossible to obtain two affine transformations $\mathcal{S}$ and $\mathcal{T}$, as well as a central map $\mathcal{Q}$ [Pat96], that belong to a special class of nonlinear polynomial systems, whose composition generates a mapping $\mathcal{P}^{'}$ that matches the transformation given by the public key $\mathcal{P}^{'} = \mathcal{S} \circ \mathcal{Q} \circ \mathcal{T} = \mathcal{P}$. The cryptanalysis techniques that consists in trying to reproduce the structure of the private key are called Structural attacks.

2.2.3 Strengths and Limitations

Besides being supposedly resistant against quantum computer attacks, cryptosystems based on multivariate polynomials, the Rainbow authors presents a series of advantageous features in the supporting documentation[DPSY19] that supposedly allow their efficient implementation in software, which directly turns them into potential solutions for small embedded devices security.

- **Speed and Signature Size**
  MPKC's are much faster in comparison to approaches like ECC or RSA, offering high speed signature generation. Furthermore, their size is about a few hundreds of bits, much less than other post-quantum signature approaches.

- **Simple Arithmetic**
  The operations that are performed in multivariate cryptosystems are based on reduced size finite field arithmetic involved in matrix products and linear system solving. Thus, MPKC's can be efficiently implemented on devices without an integrated cryptographic co-processor.

- **Multiproblem-based Security**
  Most cryptosystems used nowadays base their security on integer factorization or discrete logarithms. If anybody could find a way of solving these mathematical problems efficiently, all the encrypted data between communicating parties will be threatened. Using security principles that differ from the ones commonly employed nowadays, relieves the risk of a cryptanalytic success.

On the other hand, two main drawbacks are introduced also in [DPSY19], which limit the implementation feasibility of this post-quantum approach on memory-constrained devices.

- **Key Size**
  The size of the keys is much larger compared to the ones used in traditional approaches. In case of the public keys, huge quantities of memory are needed to store them, ranging from tenths to hundreds of kBytes. When it comes to private keys, even being smaller, they are still not able to fit in devices with memory limitations. Furthermore, the number of clock cycles elapsed for generating the keys is much higher than in the signature generation and verification procedures.

- **Parameter Set Choice**
  The security level offered by any cryptosystem mainly depends on the parameters that are chosen to define its structure. Within the context of multivariate cryptography, this is still a blurry aspect, as it is not completely defined the correlation between the security and efficiency of a scheme, depending on the selected set of parameters.

The main challenge of MPKC's is to reduce the amount of memory needed to store its keys. When an application has to verify a message, it has to store the public key, which becomes an infeasible task for those platforms that are limited in memory. If this application also needs a method for generating digital signatures, the secret key has to be kept in memory too, which despite being smaller than the public key, is still large enough for small embedded system implementations.
However, while research remains focused on the reduction of the generated keys, there are still other exploitable advantages over traditional approaches, such as the high speed algorithms thanks to the simple arithmetic employed, unlike RSA or ECC for example, which require hundreds of bits to represent each operand and therefore increase the time needed to compute the result of the operations.

3 Finite Field Theory

3.1 Finite Field Algebra

A field is an abstraction of a set of elements with mathematical operations defined over them, so that calculations can be made between the numbers so that the result stays always inside the field. These operations should satisfy the field axioms, such as associativity, commutativity and distributivity. It is also needed to have both an additive and multiplicative inverse. When the field has a finite number of elements is called Finite Field or Galois Fields, in honor of Évariste Galois, creator of the concept.
The order of a the field is the number of elements that compose the field. A Galois field $\mathbb{F}$ of order q exists, if and only if q is a prime power, such as $q = p^m$ where p is a prime number known as the characteristic of $\mathbb{F}$, and m is a positive integer. The name of the fields changes according to the value that takes m. It can be distinguished between prime fields when $m = 1$ and extension fields when $m \geq 2$. Finite fields are unique for a characteristic q, meaning that despite the possibility of having a different representation of its elements, they will be structurally the same. For this reason, two finite fields with characteristic q are called isomorphic and denoted as $\mathbb{F}_q$.

- **Prime Fields**
  Being p a prime number, integers modulo p consisting of the integers $\{0, 1, 2, ..., p-1\}$ with addition and multiplication operations performed modulo p, form a finite field of order p.

- **Binary Fields**
  Finite fields of order 2^m are called binary fields. One way to construct them is using a polynomial basis representation, where the elements of $\mathbb{F}_{2^m}$ are the polynomials of degree at most $m - 1$, whose coefficients belong to $F_2 = \{0, 1\}$.

  $$\mathbb{F}_{2^m} = \{a_{m-1}z_{m-1} + a_{m-2}z_{m-2} + \cdots + a_1 z + a_0 : a_k \in \{0, 1\}\}.$$

  Addition of field elements is computed as the usual addition of polynomials, with coefficient arithmetic performed modulo 2. Multiplication is performed modulo an irreducible polynomial $f(z)$ degree m. The structure of the irreducible polynomial has a direct impact on the efficiency of the arithmetic opeartions where reduction is needed. In fact,

choosing third and fifth degree polynomials are the best alternative for this purpose.

Some reasons why finite fields are employed in many cryptographic algorithms is because $GF(2^m)$ all the operations carry-less, there are is no rounding performed and the word length is constant [BDKR03].
Moreover, the structure of the field can be reformulated where we can consider displaying a k-bit input as m contiguous n-bit groups which corresponds to instructions in SIMD architectures. These principles offer the possibility to efficiently implement finite field arithmetic operations.

- **Extension Fields**
  The polynomial basis representation can be generalized to any extension field. Let p be a prime and $m \geq 2$. With $\mathbb{F}_p[z]$ denoting the set of all polynomials in the variable z whose coefficients belong to $\mathbb{F}_p$. The irreducible polynomial $f(z)$ is chosen to be of degree m in $\mathbb{F}_p[z]$. The coefficients that compose $\mathbb{F}_{p^m}$ are the polynomials in $\mathbb{F}_p[z]$ of degree at most $m-1$.

$$\mathbb{F}_{p^m} = \{a_{m-1}z_{m-1} + a_{m-2}z_{m-2} + \cdots + a_1 z + a_0 : a_k \in \mathbb{F}_p\}$$

3.2 Tower Field Representation

If it is possible to decompose an integer N as the product of l and m, it is possible to derive a representation of $GF(2^N)$ over $GF(2^l)$. This is because, even though the form of the elements differs between $GF((2^l)^m)$ and $GF(2^N)$, they define the same finite field.
The elements of $GF(2^N)$ are polynomials of degree at most $N-1$ whose coefficients are in $GF(2)$ = {0, 1}, and the elements of $GF((2^l)^m))$ are polynomials whose coefficients are in $GF(2^l)$ of degree at most $m-1$. The tower field can be constructed with as much layers as possible, depending on the power representation of the order. Modelling a field following this approach receives the name of tower (or composite) field representation.

Instead of using a base $\mathfrak{B}_1 = \{1, \alpha, \alpha^2, ..., \alpha^{N-1}\}$ and generating the coefficients of the field as $A = \sum_{i=0}^{N-1} a_i \alpha^i$, where a_i are the elements from $GF(2)$, we use the alternative base $\mathfrak{B}_2 = \{1, \beta, \beta^2, ..., \beta^{m-1}\}$ and the generation equation turns into $B = \sum_{i=0}^{m-1} b_i \beta^i$ with b_i as the elements of $GF(2^l)$. Employing this technique provides the capability of modelling the coefficients of a finite field, using a lower amount of elements equal to the order of the field below in the tower structure. Moreover, downgrading the order of the field translates into a fewer quantity of bits to represent the coefficients.

3.3 Karatsuba-Ofman Method

The efficient implementation of finite field arithmetic is key to the development of cryptosystems whose underlying mathematical principles are based on arithmetic operations between the elements of a field with a finite number of elements.

The Karatsuba-Ofman methodology provides an alternative representation to multiplications in order to reduce the number of computations needed to obtain the final result [GCL92]. The algorithm relies on the "divide-and-conquer" principle. It consists in solving partial problems separately and combining the solutions to obtain a joint result. In order to multiply two n-bit integers using the "schoolbook" method, considered the slowest, m^2 multiplications and $(m-1)^2$ additions are required. By means of Karatsuba-Ofman algorithm applied recursively, the number of operations is reduced to $n^{log_2 3}$ multiplications and $6n^{log_2 3} - 8n + 2$ additions [Eyu15]. Since the complexity of addition and subtraction is linear $(\mathcal{O}(n))$, this reformulation is justifiable. In order to employ this approach, the number of bits of each element has to be a power of 2. Otherwise, they are padded with zeros.

Two integers from a given finite field are represented as polynomials $A(x)$ and $B(x)$ of degree $n-1$. The goal is to obtain a resulting polynomial of degree at most $2n-2$ given by $C(x) = A(x)B(x)$. The Karatsuba-Ofman alternative model is shown in the next equations. Every step halves the number of bits required to represent the integers in the equations by using the upper and lower parts of each of the operands (A_1, A_0, B_1, B_0):

$$A(x) = A_1 x^{\frac{n}{2}} + A_0 = (x^{\frac{n}{2}-1} a_{n-1} + \cdots + a_{\frac{n}{2}}) x^{\frac{n}{2}} + (x^{\frac{n}{2}-1} a_{\frac{n}{2}-1} + \cdots + a_0)$$
$$B(x) = B_1 x^{\frac{n}{2}} + B_0 = (x^{\frac{n}{2}-1} b_{n-1} + \cdots + b_{\frac{n}{2}}) x^{\frac{n}{2}} + (x^{\frac{n}{2}-1} b_{\frac{n}{2}-1} + \cdots + b_0)$$

Multiplying the two polynomials $A(x)$ and $B(x)$ expressed in this form, gives the following result

$$C(x) = A(x)B(x) = A_0 B_0 + x^{\frac{n}{2}}[(A_0 + A_1)(B_0 + B_1) - A_0 B_0 - A_1 B_1] + x^n A_1 B_1$$

This methodology terminates after $log_2 n$ steps when is applied recursively, until the point where only constant monomials compose the polynomial equations.

4 Rainbow Signature Scheme

4.1 Unbalanced Oil & Vinegar Polynomials

The Oil and Vinegar and later Unbalanced Oil and Vinegar are schemes employed in multivariate cryptosystems for generating digital signatures. The trapdoor is achieved not by using field extensions but giving the polynomials a special algebraic structure.

The name of this approach comes from the distinction of the variables that is made over two different groups, the Oil and the Vinegar. The unknowns of each group are mixed in the polynomials of the central map. Having a balanced or unbalanced scheme, depends on the number of variables from each group present in the equations. The unbalanced case was born after the balanced approach was broken, after exploiting the possibility of building oil and vinegar polynomials independently, because of the number of variables from each group in the construction ($v = o$). To prevent this attack, the unbalanced approach was developed where a different number of unknowns is chosen from each group ($v > o$) [KPG99].

The number of unknowns is defined by $n = o + v$. The variable sets are divided into vinegar $V = \{x_1, ..., x_v\}$ and oil $O = \{x_{v+1}, ..., x_n\}$. The number of polynomials in the central map equals the number of oil variables $m = o = n - v$.

The structure of a UOV polynomial equation is given by

$$f^{(m)}(x_1, ...x_n) := \sum_{i=1}^{v}\sum_{j=1}^{v} \alpha_{i,j}^{(m)} x_i x_j + \sum_{i=1}^{v}\sum_{j=v+1}^{n} \beta_{i,j}^{(m)} x_i x_j + \sum_{i=1}^{n} \gamma_i^{(m)} x_i + \delta^{(m)}$$

where $\alpha_{i,j}^{(m)}$, $\beta_{i,j}^{(m)}$, $\gamma_{i,j}^{(m)}$ and $\delta_{i,j}^{(m)}$ are chosen from a finite field of size q ($\mathbb{F}_q$).

The most significant algebraic property of this scheme is that the oil and vinegar variables are not fully mixed, which means that there will not exist quadratic terms between two variables from the oil set in the equations of the central map. This principle makes the central map $\mathcal{Q}$ easily invertible. This is not the case for the public map $\mathcal{P}$, as the affine transformation $\mathcal{S}$ fully mixes all the variables.

In order to sign a given message x = $(x_1, ..., x_o)$, a vector w = $(w_1, ..., x_n)$ that satisfies $x = \mathcal{P}(w)$ the public key polynomial equation system has to be inverted, which is a possible task if information about the secret key is available.
To invert the central map, random values are chosen for the vinegar variables $x_1, ..., x_v$, obtaining o linear equations in the oil variables $x_{v+1}, ..., x_n$.
This linear system is, with high probability, solvable by Gaussian Elimination [Laz83]. In case that the system has no solution, different random vinegar variables are chosen and the process is restarted.

To generate the public key of a cryptosystem based on UOV polynomials, only one affine map is needed to obfuscate the polynomials of the central map. The reason that explains this practice comes from the fact that the trapdoor remains unaltered after composing $\mathcal{S}$ with $\mathcal{Q}$, and thus does not contribute to the security of the scheme. Therefore, $\mathcal{S}$ is dropped and the public key is constructed as $\mathcal{P} = \mathcal{Q} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^o$.

The main drawback of the Unbalanced Oil & Vinegar approach is the size of the keys and signatures generated, which becomes infeasible for some practical applications. In order to reduce the size of the keys without degrading the security level of the scheme, **Rainbow** was first proposed in 2005 by J. Ding and D. Schmidt in [DS05].

4.2 Rainbow Construction Principle

Rainbow is a signature scheme based on multivariate cryptography whose central map is constructed stacking various Unbalanced Oil & Vinegar-based layers [DS05, DYC$^+$08]. The multi-layered structure is designed hierarchically, in order to reuse already computed coefficients in upcoming layers and create dependencies between them, to improve the overall efficiency of the algorithm. An exhaustive analysis of Rainbow's security can be found in [DYC$^+$08].

Let V be the set of integers $\{1, v_1, v_2, ..., v_u, n\}$. For a Rainbow signature scheme with u stacked layers, two variable sets are defined to design the polynomials of the central map, where $l = 1, ..., u-1$; In the one hand, V_l = {1, ..., v_l} with v_l elements and in the other hand O_l = $V_{l+1} \setminus V_l$ = $v_l + 1, ..., v_{l+1}$ with $o_l = v_{l+1} - v_l$ elements, as it shown in Figure 4.1 for a 2-layer Rainbow construction. Following this construction manner, the vinegar variable set for each layer satisfies $V_1 \subset V_2 \subset ... \subset V_u = V$.

The central map $\mathcal{Q}$ is represented as a set of $m = n - v_1$ polynomial equations $f^{(v_1+1)}, ..., f^{(n)}$ that present the following form:

$$f^{(k)}(x_1, ...x_n) := \sum_{i,j \in V_l} \alpha_{i,j}^{(k)} x_i x_j + \sum_{i \in V_l, j \in O_l} \beta_{i,j}^{(k)} x_i x_j + \sum_{i \in V_l \bigcup O_l} \gamma_i^{(k)} x_i + \delta^{(k)}$$

where $\alpha_{i,j}^{(k)}$, $\beta_{i,j}^{(k)}$, $\gamma_{i,j}^{(k)}$ and $\delta_{i,j}^{(k)}$ are chosen from $\mathbb{F}_q$ and l is an integer such that $k \in O_l$.

The polynomials of the central map are distributed between the different layers. As in UOV, there are no crossed terms between $x_i x_j$ where $i, j \in O_l$.

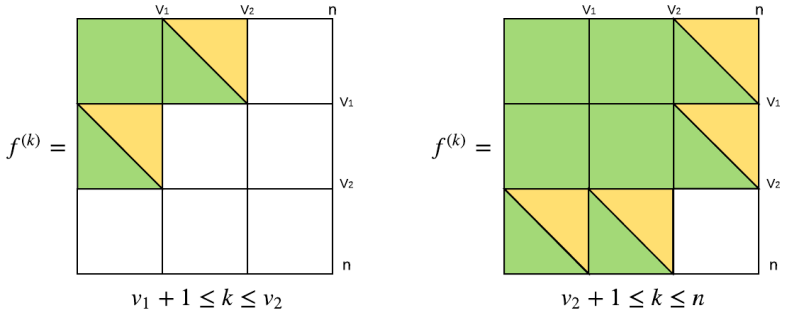

Figure 4.1: Quadratic distribution of the central map in each of the two-layer Rainbow scheme. Green parts correspond the $v \times v$ variables, mixed yellow-green to $v \times o$ crossed variables and white to zero entries.

The **public key** $\mathcal{P}$ of the Rainbow scheme consists in a system of $n - v_1$ polynomial equations, composed by a central map $\mathcal{Q}$ with the described form in 4.1, and two affine transformations $\mathcal{S}$ and $\mathcal{T}$. In order to mix the variables from the polynomials of the central map, we compute $\mathcal{S} \circ \mathcal{Q} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^m$. The central map $\mathcal{Q}$ can be displayed as a matrix-vector product.

$$\mathcal{Q} = \begin{bmatrix} x_1 & \cdots & x_n \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1v} & x_{1(v+1)} & \cdots & x_{1n} \\ \vdots & & \vdots & \vdots & & \vdots \\ x_{v1} & \cdots & x_{nv} & x_{n(v+1)} & \cdots & x_{vn} \\ x_{(v+1)1} & \cdots & x_{(v+1)v} & 0 & \cdots & 0 \\ \vdots & & \vdots & \vdots & & \vdots \\ x_{n1} & \cdots & x_{nv} & 0 & \cdots & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

The **private key** consists of the three mappings $\mathcal{S}, \mathcal{Q}, \mathcal{T}$. In this case, it is required a second affine transformation to mix the polynomials from $\mathcal{S} \circ \mathcal{Q}$ between them, unlike the UOV approach, as the trapdoor $\mathcal{Q}$ does not get altered when it is composed with $\mathcal{T}$, and therefore its contribution to the scheme's security is null.

Rainbow is considered a booster stage to the Unbalanced Oil & Vinegar approach, where the size of the keys and signatures is notably shortened. Because of the hierarchical stack building manner, the parameters that define the scheme $(v_1, o_1, o_2, ...)$ can be chosen smaller than in the simple UOV construction, leading to shorter keys and smaller message to signature ratios. The number of coefficients from $\mathbb{F}_q$ that compose the public and the private keys of an u-layered Rainbow scheme is given by the following equations.

$$size_{p_k} = m \cdot \frac{(n+1)\cdot(n+2)}{2}$$

$$size_{s_k} = m\cdot(m+1) + n\cdot(n+1) + \sum_{i=1}^{u} o_l \cdot (\frac{v_l\cdot(v_l+1)}{2} + v_l \cdot o_l + v_{l+1} + 1)$$

Signing Methodology.
To sign a message, the trapdoor function has to be inverted, which is an easy task if there is information available about the mappings that compose the secret key. Reverting the affine transformations $\mathcal{S}$ and $\mathcal{T}$ is simple, while performing this task for the central map is slightly more complex. Below is explained the signing procedure for a two-layer Rainbow scheme.

In Figure 4.2, it is shown the polynomial structure of the central map. The steps followed to sign a message are described in the following lines and illustrated in 4.3 and 4.4. The colors employed in the plots are organized as follows : blue for quadratic terms, green for linear terms in V_1, yellow for linear terms in O_1, red for linear terms in O_2 and white for constant terms.

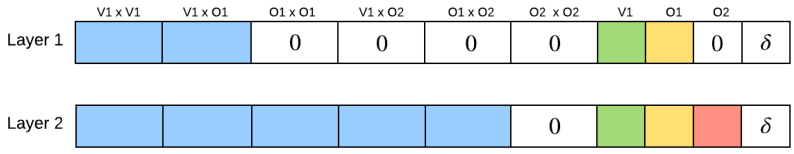

Figure 4.2: Central map of the Rainbow scheme

- Random values for the vinegar variables $x_1, ..., x_{v_1}$ are chosen and substituted into the central map polynomials $(f^{(v_1+1)}, ..., f^{(n)})$ obtaining a system of o_1 linear equations, given by the polynomials of the first layer $k \in O_1$.

- Oil variables $x_{v_1+1}, ..., x_{v_2}$ are found by solving the o_1 linear equations from the first layer. After that, these are substituted into the equations of the second layer.

- The central map turns into an equation system of o_2 linear polynomials of the second layer $k \in O_2$ over the oil variables $x_{v_2+1}, ..., x_{v_n}$ which can be solved with Gaussian Elimination.

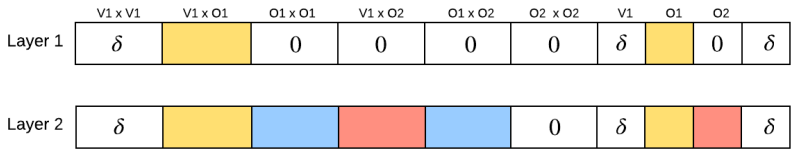

Figure 4.3: First step of the Rainbow signature procedure

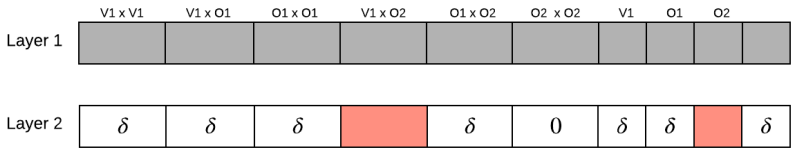

Figure 4.4: Second step of the Rainbow signature procedure

In case that one of the linear equation systems does not have a solution, new vinegar variables $x_1, ..., x_{v_1}$ from the first layer are chosen and the process is repeated again.

4.3 Parameter Sets

Choosing a certain set of parameters to build a cryptographic scheme defines its level of security as well as the capacity to implement it efficiently. Rainbow offers two degrees of freedom to build its structure. In the one hand, the size q of the finite field ($\mathbb{F}_q$) where the numerical units used in this approach are selected from. On the other hand, the parameters corresponding to the number of layers u and the size of both the oil and vinegar variables sets of each layer $(v_1, o_1, \cdots, o_u)$.

Regarding the choice of the field's size, it is a common practice to select 2^8 as its size given the efficient implementations that can be implemented when working with 8-bit variables. Multiples of this size are also chosen, even though further processing of the elements of the field, like splitting or merging is required, for implementing fast arithmetic operations.

About the choice of this parameters in order to design the structure of each layer, it can be claimed that, even though Rainbow can be defined for an arbitrary number of stages u, choosing $u = 2$ offers a more efficient scheme that generates shorter keys at the same level of security offered by a scheme designed with other values of u. Choosing $u > 2$ leads to a small improvement in performance terms while increasing notably the size of the keys to reach the same level of security. It is a common practice to choose the size of the layers to be equal ($o_1 = o_2 = \cdots = o_u$) due to implementation ease and efficiency-related reasons.

Furthermore, the number of vinegar variables is chosen to be two ($v = 2o$) or three ($v = 3o$) times the number of oil variables, given that $v \leq o$ and $v >> o$ compromises the security of the cryptosystem. All these principles are presented and justified in [DS05].

Diverse research has been made to determine the suitable parameters depending on the user's interest by studying the correlation between the selection of values and the level of security offered by the scheme. In fact, on the supporting documentation of Rainbow [DPSY19], the authors claim four statements that relate the choice of concrete parameter sets with the vulnerabilities that are generated over the security of the cryptosystem which can be exploited by different cryptanalysis techniques. These remarks are listed below.

- If the number of equations ($o_1 + o_2$) that form the public key mapping is not large enough, the scheme becomes vulnerable to **direct attacks**.
- The value of v_1, corresponding to the number of vinegar variables of the first layer, has to be large enough to prevent the **Rainbow-Band-Separation attack**.
- Choosing the number of oil variables in the second layer o_2 too high will make the scheme vulnerable to the **UOV attack**.
- A small value of o_2 can get the security of the scheme compromised when facing the **HighRank attack**.

4.4 Algorithms

The Rainbow signature scheme provides three different available cryptographic functionalities. The first is **Key Generation**, which returns a duple composed by the public and the secret key whose size will depend on the parameters chosen to design the structure of the cryptosystem. The second is **Signature Generation**, which implies the encryption, with the secret key, of the hash value generated from the message to sign, in order to obtain its digest. The last procedure is **Signature Verification**, which authenticates the actual sender of the signature to verify, by comparing both the hash of the message and the hash of the decrypted signature using the public key.
Along the following pseudo-code description of the algorithms, different functions are being called. Instead of also showing their whole implementation, we directly provide a brief description for each of them in Table 4.1.

Functions	Output
$\texttt{isInvertible}(M)$	True or False if the input matrix has an inverse
$\texttt{Aff}(M, c)$	Affine transformation $M \cdot x + c$
$\texttt{Matrix}(q, s1, s2)$	$s1 \times s2$ matrix with random coefficients from $\mathbb{F}_q$
$\texttt{CentralMap}(q, v_1, o_1, o_2)$	Rainbow central map according to the input parameters
$\texttt{Aff}^{-1}(\widehat{f^{(v_1+1)}}, ..., \widehat{f^{(n)}})$	Affine inverse of the system $(\widehat{f^{(v_1+1)}}, ..., \widehat{f^{(n)}})$
$\texttt{Gauss}(\widehat{f^{(v_2+1,...,n)}} = (x_{v_2+1}, ..., x_n)$	Random solution of the input equation system

Table 4.1: Description of the functions implicitly called in the KeyGen, Sign and Verify algorithms pseudo-code.

4.4.1 Key Generation

The **private key** consists in two invertible affine maps $\mathcal{S}(M_S, c_S) : \mathbb{F}^m \to \mathbb{F}^m$ and $\mathcal{T}(M_T, c_T) : \mathbb{F}^n \to \mathbb{F}^n$, as well as a quadratic central map $\mathcal{Q} : \mathbb{F}^n \to \mathbb{F}^m$. Random values from a given finite field of size q are selected to fill the coefficient matrices and vectors of each mapping. The **public key** is generated by computing the composition $\mathcal{P} = \mathcal{S} \circ \mathcal{Q} \circ \mathcal{T} : \mathbb{F}^n \to \mathbb{F}^m$.
The affine transformation $\mathcal{S}$ mixes the variables of in the polynomials of the central map. Otherwise, the mapping $\mathcal{T}$ is responsible for mixing the polynomials of the composition between them. As mentioned in 4.2, after this process the public map $\mathcal{P}$ looks like a random polynomial system.

Algorithm 1 Rainbow Key Generation

Input: Rainbow parameters (q, v_1, o_1, o_2), length of salt l
Output: Rainbow key pair (p_k, s_k)

$m \leftarrow o_1 + o_2$
$n \leftarrow m + v_1$
while $\texttt{isInvertible}(M_s)$ == **FALSE do**
    $M_S \leftarrow \texttt{Matrix}(q, m, n)$
$c_S \leftarrow \texttt{rand}(\mathbb{F}^n)$
$\mathcal{S} \leftarrow \texttt{Aff}(M_S, c_S)$
while $\texttt{isInvertible}(M_T)$ == **FALSE do**
    $M_T \leftarrow \texttt{Matrix}(q, n, n)$
$c_T \leftarrow \texttt{rand}(\mathbb{F}^n)$
$\mathcal{T} \leftarrow \texttt{Aff}(M_T, c_T)$
$\mathcal{Q} \leftarrow \texttt{CentralMap}(q, v_1, o_1, o_2)$
$\mathcal{P} \leftarrow \mathcal{S} \circ \mathcal{Q} \circ \mathcal{T}$
$p_k \leftarrow p_k, l$
$s_k \leftarrow s_k, l$
return (p_k, s_k)

4.4.2 Message Signature

To sign a message d, **x** = $\mathcal{S}^{-1}$(**h**) $\in \mathbb{F}^m$ is computed, where **h**=$\mathcal{H}(d) \in \mathbb{F}^m$ and $\mathcal{H}$:{0,1}$\rightarrow \mathbb{F}^m$. After that, a pre-image **y** $\in \mathbb{F}^n$ of the central map $\mathcal{Q}$ is found as explained at the end of Section 4.2. Finally $\mathcal{T}^{-1}$(**y**) is computed to obtain the signature **z** $\in \mathbb{F}^n$.

Algorithm 2 Rainbow Signature Generation

Input: message d, private key $s_k = (InvS, c_S, \mathcal{F}, InvT, c_T)$, length l of the salt
Output: signature s = (**z**,r) $\in \mathbb{F}^n \times \{0,1\}^l$ such that $\mathcal{P}$(**s**) = $\mathcal{H}(\mathcal{H}(d) \parallel r)$

while `isInvertible`$(\widehat{F})$ == **FALSE do**
    $y_1, ..., y_{v_1} \leftarrow$ `rand`$(\mathbb{F}^n)$
    $\widehat{f}^{(v_1+1)}, ..., \widehat{f}^{(n)} \leftarrow f^{(v_1+1)}(y_1, ..., y_{v_1}), ..., f^{(n)}(y_1, ..., y_{v_1})$
    $(\widehat{F}, c_F) \leftarrow$ `Aff`$^{-1}(\widehat{f}^{(v_1+1)}, ..., \widehat{f}^{(n)})$
$InvF = \widehat{F}^{-1}$
while t == **FALSE do**
    $r \leftarrow \{0,1\}^l$
    h $\leftarrow \mathcal{H}(\mathcal{H}(d) \parallel r)$
    x $\leftarrow InvS \cdot$ (**h** $- c_s$)
    $y_{v_1+1}, ..., y_{v_2} \leftarrow InvF \cdot ((x_{v_1+1}, ..., x_{v_2}) - c_F)$
    $\widehat{f}^{(v_2+1)}, ..., \widehat{f}^{(n)} \leftarrow f^{(v_2+1)}(y_{v_1+1}, ..., y_{v_2}), ..., f^{(n)}(y_{v_1+1}, ..., y_{v_2})$
    $t, (y_{v_2+1}, ..., y_n \leftarrow$ `Gauss`$(\widehat{f}^{(v_2+1)} = x_{v_2+1}, ..., \widehat{f}^{(n)} = x_n)$
z = $InvT \cdot$ (**y** $- c_T$)
$s \leftarrow$ (**z**, r)
return (s)

4.4.3 Signature Verification

In order to check the authenticity between a signature **z** and a given document d, one obtains the hash value **h** = $\mathcal{H}(d) \in \mathbb{F}^m$ in first place and computes **h'** = $\mathcal{P}$(**z**) $\in \mathbb{F}^m$ in second place. In case that both **h** and **h'** are equal, the signature **z** is verified successfully.

Algorithm 3 Rainbow Signature Verification

Input: message d, signature s = (**z**,r) $\in \mathbb{F}^n \times \{0,1\}^l$
Output: Boolean True or False

h $\leftarrow \mathcal{H}(\mathcal{H}(d) \parallel r)$
h' = $\mathcal{P}$(**s**)
if h' == h then then
    return TRUE
else
    return FALSE

5 Rainbow in ARM Cortex-M4

5.1 Description of the Environment

As it has been exposed in the introduction of this project, the purpose of this work is to evaluate the performance of Rainbow in a small embedded microprocessor, followed by the study of optimization alternatives to improve the efficiency on our platform.
As starting point, we used the Round 1 [rou17] and Round 2 [rou19] reference implementation of the Rainbow algorithms, which have been obtained from the post-quantum cryptography submissions in the CSRC proposed by NIST.

An ARM Cortex-M4 on a STM32 Nucleo L4R5ZI board (STMicroelectronics) has been used as the target device. The MCU presents a 32-bit architecture and is provided with a FPU, as well as a ART Accelerator, MPU and DSP instructions. The clock frequency of the core is 120 MHz and the memory space is shared between 2 Mbytes of Flash and 640 Kbytes of SRAM.

The code is written in C. To compile and optimize the code, we used the version 6.3.1 of the gcc-arm-none-eabi cross-compiler, obtained from the GNU Embedded Toolchain for ARM[1]. In order to interact with the MCU, without the need of reaching register level to access certain features of the board, HAL drivers have been used for this purpose. The device has an integrated USB to UART bridge. Therefore, the communication between the PC and the MCU is straightforward with the corresponding HAL driver.

The linker script defines where are stored the different sections of the code, either in the SRAM or in the Flash memory. This has to be taken into account to keep the size of the stack and the heap under control and to be aware of how much memory left is available, which is required to evaluate the memory footprint that has any program/function during its execution. The program instructions (.text) are stored in the Flash, as well as constant data. The SRAM contains other sections such as uninitialized (.bss) and initialized (.data) data. The remaining unused space in the SRAM is left for stack and heap usage.

[1] `https://launchpad.net/ubuntu/+source/gcc-arm-none-eabi`

5.2 Implementation Details

The schemes that NIST intends to standardize should enable existentially unforgeable digital signatures concerning an adaptive chosen message attack. This property is called **EUF-CMA** security. In order to fulfill this security model in both Round 1 and Round 2 reference implementations of the algorithm, a binary vector r called salt is introduced in the scheme's cryptographic procedures, in order to ensure that no attacker is able to forge any hash-signature pair. The length of this vector is chosen such that 2^{64} plaintexts can be signed with a single key pair.

Below are described the modifications that are applied to the standard methodology of key generation, message signature and signature verification, concerning the security level that has to be obtained.

- **Rainbow Key Generation** [Algorithm 1]
  An integer l is chosen as the length of the salt vector and later appended both to the public and private key.

- **Rainbow Signature Generation** [Algorithm 2]
  Vinegar variables are randomly chosen $(x_1, ..., x_{v_1})$. After that, a random salt vector of length l is chosen $r \in \{0,1\}^l$ and the standard signature procedure is executed for $\mathcal{H}(\mathcal{H}(d) \parallel r)$ to obtain a signature s = (**z** $\parallel$ r). If any linear equation system has no solution, a new salt vector r is chosen and the process is restarted .

- **Rainbow Signature Verification** [Algorithm 3]
  The algorithm uses $\mathcal{H}(\mathcal{H}(d) \parallel r)$ instead of $\mathcal{H}(d)$, to check the equality to $\mathcal{P}(z)$ and thus verify if the signature **z** corresponds to the message d.

The implementation uses SHA-2 functions to hash messages during the signature and verification procedures. This family provides four different options: SHA224, SHA256, SHA384 and SHA512 which output hash values of length equal to 224, 256, 384 and 512 bits respectively.
Random numbers are generated both during the keys and signature generation. For that purpose, the submitted reference implementation uses functions from the OpenSSL library. A secret seed is required to generate random integers in the same way a pseudo-random number generator would do.

5.2.1 Representation of Finite Field Elements

The reference implementation of Round 1 and Round 2 stores the coefficients from each finite field as polynomials following a Tower field structure, in order to obtain time-constancy and prevent side-channel attacks.

The tower field employed in the Rainbow implementation is constructed as follows

$$GF(4) \coloneqq GF(2)\ [x]\ /\ x^2 + x + 1$$
$$GF(16) \coloneqq GF(4)\ [y]\ /\ y^2 + y + x$$
$$GF(256) \coloneqq GF(16)\ [z]\ /\ z^2 + z + xy$$

The elements from $GF(4)$ are $\{0, 1, x, x-1\}$, where the coefficient set consists in {0,1} and the base of the field is $\mathfrak{B}_4 = \{1, x\}$. To generate $GF(16)$, the coefficients are the elements from $GF(4)$ and the base is $\mathfrak{B}_{16} = \{1, x, y, xy\}$. Finally, $GF(256)$ is assembled combining of the coefficients from $GF(16)$ and $\mathfrak{B}_{256} = \{1, x, y, xy, z, xz, yz, xyz\}$. The elements of the finite field are generated by creating different linear combinations between the elements of the base and the coefficients from $GF(2)$.

- $GF(16)$
  Two bits are needed to store each $GF(4)$ coefficient of the linear polynomial ($\alpha \cdot x + \beta$) representing the elements of GF(16). Therefore, 4 bits are needed to store an element of $GF(16)$. Coefficients from GF(2) are stored in 1 bit. The linear and the constant term of the polynomials are stored in the most and least significant bits respectively. Elements from this field are usually packed in pairs into the same 8-bit integer which is the smallest unit available in C language.

- $GF(256)$
  One byte is enough to store a GF(256) element, as the coefficients of the linear polynomial require 4 bits each, given that they belong to $GF(16)$. From this point, the values are stored as mentioned above down to $GF(2)$.

5.2.2 Key Storage

Before implementing a MPKC's on a micro controller, it is important to think about an efficient way of storing the keys in memory in order to speed the process of reading them during execution. As explained in the last chapter, the mappings in any multivariate quadratic construction are represented by the coefficients of their polynomials. Taking into account that random accesses to memory produces a big amount of overhead while calculating the address every time, the best alternative consists in storing the coefficients serially, where

only increments are required. Following this methodology, there are no gaps in memory which is also a memory efficient technique.

Public Key

The public key of the Rainbow scheme is based on a system of m multivariate quadratic polynomials in n variables. The equations take the following form

$$y_k = q_{1,1,k}x_1x_1 + q_{2,1,k}x_2x_1 + q_{2,2,k}x_2x_2 + q_{3,1,k}x_3x_1 + \cdots + l_{1,k}x_1 + l_{2,k}x_2 + \cdots + l_{n,k}x_n + c_k$$

where $q_{i,j,k}$ is the quadratic coefficient of x_ix_j, $l_{i,k}$ the coefficient of the linear monomial x_i and c_k the constant term of the polynomial y_k ($1 \leq j \leq i \leq n, 1 \leq k \leq m$).

In order to arrange the public key in the same array, we store the linear terms first, followed by the quadratic and the constant terms. The following sequence exemplifies how does the MQ polynomials look in memory.

$$[l_{1,1}, l_{1,2}, \cdots, l_{1,m}, l_{2,1}, \cdots, l_{n,m}, q_{1,1,1}, q_{1,1,2}, \cdots, q_{1,1,m}, q_{2,1,1}, \cdots, q_{n,n,m}, c_1, \cdots, c_m]$$

Secret Key

The private key, consisting of two affine transformations and a multi-layer UOV central map, are stored in the order $\mathcal{S}, \mathcal{Q}$ and $\mathcal{T}$.

The affine maps $\mathcal{S} : \mathbb{F}^m \to \mathbb{F}^m$ and $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$ are comprised by a $m \times m$ matrix and a $m \times 1$ vector for $\mathcal{S}$ and a $n \times n$ matrix and a $n \times 1$ vector for $\mathcal{T}$.

$$\mathcal{T}(\mathbf{x}) = \begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ & \vdots & \ddots & \\ t_{n1} & t_{n2} & \cdots & t_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} c_1 \\ \vdots \\ c_n \end{bmatrix}$$

The matrix M is stored first in column-major form. The components of the vector c are appended next to the last element of the matrix. Therefore, the affine transformation $\mathcal{T} : \mathbb{F}^n \to \mathbb{F}^n$ are displayed in memory as the following array.

$$[t_{11}, t_{21}, \cdots, t_{n1}, t_{12}, \cdots, t_{nn}, c_1, \cdots, c_n]$$

The same technique is used for storing the affine map $\mathcal{S} : \mathbb{F}^m \rightarrow \mathbb{F}^m$.

The central map $\mathcal{Q}$ is composed by two layers. In the first stage, there is $V_1 = \{1, ..., v_1\}$ and $O_1 = \{v_1 + 1, ..., v_1 + o_1\}$ and in the second $V_2 = \{1, ..., v_2 = v_1 + o_1\}$ and $O_2 = \{v_2 + 1, ..., n = v_2 + o_2\}$. Each layer is stored independently following the same manner.

The coefficients that describe the equations of the central map are divided into 3 different groups depending on the type of variable that they multiply: **vv**, **vo** and **o-linear**. They are stored in memory in the order o-linear, followed by vo and vv.

vv : The vv part is a multivariate quadratic equation system that is composed, not only by the quadratic vinegar crossed terms ($v \times v$), but also by the linear coefficients of the vinegar variables and the constant terms. The coefficients of the vv part are stored following the same manner as the public key MQ polynomial system. The components of this group present the following form

$$\sum_{i,j \in V_l} \alpha_{i,j}^{(k)} x_i x_j + \sum_{i \in V_l} \gamma_i^{(k)} x_i + \delta^{(k)}$$

vo : The vo part contains the quadratic crossed terms between the oil and the vinegar variable set

$$\sum_{i \in V_l, j \in O_l} \beta_{i,j}^{(k)} x_i x_j \rightarrow \begin{bmatrix} \beta_{11}^{(k)} & \beta_{21}^{(k)} & \cdots & \beta_{v_1 1}^{(k)} \\ & \vdots & \ddots & \\ \beta_{1o_1}^{(k)} & \beta_{2o_1}^{(k)} & \cdots & \beta_{v_1 o_1}^{(k)} \end{bmatrix}$$

The coefficients from the vo group that correspond to the polynomials of the first layer are stored in the form of a o_1 column-major matrix of size $o_1 \times v_1$, as can be observed in the following array

$$[\beta_{11}^{(v_1+1)}, \beta_{21}^{(v_1+1)}, \cdots, \beta_{1o_1}^{(v_1+1)}, \beta_{21}^{(v_1+1)}, \cdots, \beta_{v_1 o_1}^{(v_1+1)}, \beta_{11}^{(v_1+2)}, \cdots, \beta_{v_1 o_1}^{(v_1+2)}]$$

o-linear : This group contains the linear coefficients in the oil variables of the first layer. The terms that correspond to this part of the central map take the form

$$\sum_{i \in O_l} \gamma_i^{(k)} x_i \rightarrow \begin{bmatrix} \gamma_{v_1+1}^{(v_1+1)} & \gamma_{v_1+2}^{(v_1+1)} & \cdots & \gamma_{v_2}^{(v_1+1)} \\ & \vdots & \ddots & \\ \gamma_{v_1+1}^{(v_2)} & \gamma_{v_1+2}^{(v_2)} & \cdots & \gamma_{v_2}^{(v_2)} \end{bmatrix}$$

The o-linear group of coefficients is stored in the form of a row-major matrix, as shown in the following sequence

$$[\gamma_{v_1+1}^{(v_1+1)}, \gamma_{v_1+2}^{(v_1+1)}, \cdots, \gamma_{v_2}^{(v_1+1)}, \gamma_{v_1+1}^{(v_1+2)}, \cdots, \gamma_{v_2}^{(v_2)}]$$

5.2.3 Finite Field Arithmetic

Galois Field Arithmetic plays a key role in the performance of cryptographic schemes. For this reason, it is important to look for efficient software implementations, given that only in a few fields of concrete size is possible to perform hardware-supported operations.

- **Addition**
  The reference implementation performs additions and subtractions with 8-bit integers, corresponding to elements from $GF(256)$, so no merging or splitting of has to be performed. It is the most efficient way to exploit the size of the smallest variable type provided by the integer standard library of the C language. For this reason, numbers from $GF(16)$ have to be arranged in pairs before being added to another pair of $GF(16)$ coefficients. In case of additions between $GF(256)$ integers, the operation is performed directly. The addition performed modulo 2, as an XOR binary function.

- **Multiplication**
  In the reference algorithm of both rounds, the multiplication is implemented to be time-constant, taking advantage of the tower field representation. The methodology used is the so-called Karatsuba-Ofman, which is recursively employed until the terms of the multiplication are expressed with coefficients from $GF(2)$, where the operation is entirely performed at hardware level as an AND function. Multiplications between $GF(16)$ elements are faster than $GF(256)$, as in the latter case, one more decomposition stage of the tower representation has to be undergone.

5.2.4 Comparison between Rainbow Variants

Up to this point, we have described the most relevant details in the implementation that are shared between the two submissions of Rainbow to the NIST post-quantum competition. The following lines describe the differing parts between both rounds and the impact they have over the general behaviour of the scheme. The second round submission of the algorithm is basically a variant of the first submission, that yields three different algorithms for each parameter set provided, with improved performance in terms of execution time and memory called after **Classic, Cyclic and Compressed** Rainbow.

- **Improved Key Generation**
  The three proposed schemes in Round 2 generate the pair of keys much faster, given that the interpolation method, used in Round 1, is substituted by a matrix multiplication approach, which reduces drastically the number of instructions to perform the task.

  The mappings that describe the private key $\mathcal{S}$, $\mathcal{Q}$ and $\mathcal{T}$ are restricted to an homogeneous composition which results into an homogeneous public key $\mathcal{P}$. A polynomial is called to be homegenous when its non-zero terms have the same degree. Therefore, all the Rainbow mappings are composed only by second degree polynomials, leaving out the linear and the constant terms.
  The security level of Rainbow is not compromised by this modification, given that the non-homogeneous part of the matrices does not provide further resilience to the scheme.

  Furthermore, the linear maps $\mathcal{S}$ and $\mathcal{T}$ are arranged in a special form, as shown below.

$$\mathcal{S} = \begin{bmatrix} 1_{o_1 \times o_1} & S'_{o_1 \times o_2} \\ 0_{o_1 \times o_2} & 1_{o_2 \times o_2} \end{bmatrix} \mathcal{T} = \begin{bmatrix} 1_{v_1 \times v_1} & T^{(1)}_{v_1 \times o_1} & T^{(2)}_{v_1 \times o_2} \\ 0_{o_1 \times v_1} & 1_{o_1 \times o_1} & T^{(3)}_{o_1 \times o_2} \\ 0_{o_2 \times v_1} & 0_{o_2 \times o_1} & 1_{o_2 \times o_2} \end{bmatrix}$$

- **Parameter Sets**
  From the 9 parameter sets that are proposed in Round 1 by combining $GF(16)$, $GF(31)$ or $GF(256)$ and the different options to choose from SHA-2, Round 2 reduces the number of sets available to 3. The roman numbers indicate the NIST security category that the scheme aims at. Moreover, I and II employ SHA256, III and IV SHA384 and V and VI SHA512. The letter indicates the finite field used (a for $GF(16)$, b for $GF(31)$ and c for $GF(256)$).

Set	Parameters	#Equations	#Variables
Ia ($\mathbb{F}, v_1, o_1, o_2$)	$GF(16), 32, 32, 32$	64	96
Ib ($\mathbb{F}, v_1, o_1, o_2$)	$GF(31), 36, 28, 28$	56	92
Ic ($\mathbb{F}, v_1, o_1, o_2$)	$GF(256), 40, 24, 24$	48	88
IIIb ($\mathbb{F}, v_1, o_1, o_2$)	$GF(31), 64, 32, 48$	80	144
IIIc ($\mathbb{F}, v_1, o_1, o_2$)	$GF(256), 68, 36, 36$	72	140
IVa ($\mathbb{F}, v_1, o_1, o_2$)	$GF(16), 56, 48, 48$	96	152
Vc ($\mathbb{F}, v_1, o_1, o_2$)	$GF(256), 92, 48, 48$	96	188
VIa ($\mathbb{F}, v_1, o_1, o_2$)	$GF(16), 76, 64, 64$	128	204
VIb ($\mathbb{F}, v_1, o_1, o_2$)	$GF(31), 84, 56, 56$	112	196

Table 5.1: List of the proposed parameter sets for the Round 1 submission of the Rainbow algorithm

Set	Parameters	#Equations	#Variables
Ia ($\mathbb{F}, v_1, o_1, o_2$)	$GF(16), 32, 32, 32$	64	96
IIIc ($\mathbb{F}, v_1, o_1, o_2$)	$GF(256), 68, 36, 36$	72	140
Vc ($\mathbb{F}, v_1, o_1, o_2$)	$GF(256), 92, 48, 48$	96	188

Table 5.2: List of the proposed parameter sets for the Round 2 submission of the Rainbow algorithm

- **Memory-Speed Trade-Off**
  Cyclic and compressed Rainbow variants are characterized by drastically reducing the memory usage while losing efficiency during execution. In the one hand, cyclic Rainbow is based on Petzoldt's cyclic scheme [PBB10], which allows, not only to insert cyclic matrices into the public key map but also to generate most of its parts by using a stored seed [DBM12]. This approach saves up to the 70% of the memory space employed in Round 1. In the other hand, the compressed version stores the secret key as two 256 bit seeds. The loss of efficiency is given by the fact that every time the public or secret keys have to be used in either verification or signing routines respectively, they have to be decompressed from the stored seed that generates them, leading to a slower execution.

5.3 Modifications to the Reference Implementation

In order to implement the Rainbow algorithm in the Cortex-M4, we have to get rid of most of the libraries used in the reference implementation due to the huge overhead that is generated. The strict memory constraints imposed by the MCU and the incompatibilities in the code between x86_64 and ARM compilers are the reasons that explain this practice. Fortunately, only the OpenSSL library is employed, both for document **hashing** and generating **random bytes**.

We replaced the OpenSSL library used in the given reference implementation with the mbedTLS[2], a cryptographic library with SSL and TLS capabilities for ARM microprocessors. Even though we managed to compile the mbedTLS library for our architecture, a huge amount of memory was consumed in order to handle the storage of this library, without leaving enough space for implementing the algorithm on the device.

After discussing which could be the best option for omitting the use of libraries without downgrading the overall performance, we decided to

[2] `https://tls.mbed.org`

- **Hash Functions**
  Employ the hashing functions for SHA256, SHA384 and SHA512 based on Daniel Bernstein's public domain implementation.

- **RNGeneration**
  Use the Pseudo-Random Number Generator (PRNG) integrated in the Nucleo L4R5ZI board, which uses hardware noise as the source to obtain a seed for generating aleatory numbers. Therefore, all the functions and structures related to seed generation and PRNG configuration are dismissed.
  To take advantage of the integrated PRNG, we use the HAL driver available for this purpose. More specifically, we call the function `HAL_RNG_GenerateRandomNumber`, that returns a 32-bit random number. As the original function from the reference implementation returns n random bytes, we had to adapt our method to return the demanded number of bytes correctly by performing 8-bit splits over the output of the HAL PRNG function.

- **Memory Allocation**
  Instead of using `aligned_alloc()` for dynamically allocating memory, we directly use `malloc()`, given that the C standard library for the cross-compiler arm-none-eabi-gcc does not provide this function. This change is only applied in Round 2.

5.4 Rainbow Embedded Application

For visualizing and evaluating the way key generation, signature and verification algorithms work, we embedded the Rainbow scheme in a real application.
The MCU is flashed with three different instances of the Rainbow scheme constructed from the parameter sets **Ia** and **Ic** from Round 1 and **Ia** from Round 2. The security provided by these parameter sets corresponds to level 1 of the NIST security categories for the Ia and level 2 for the Ic. The parameter sets that we evaluate in this thesis use SHA256 to generate hash values. Because of the limitations imposed by the memory of our device, we could not fit any of the other parameter set alternatives proposed.
From round 2 we only select the Ia_Classic approach, leaving out the cyclic and compressed variants from the evaluation. Furthermore, We have not included the set **Ib**, which uses $GF(31)$ as the underlying finite field, given that most part of the research papers tend to dismiss its use due to the efficiency loss that it generates.

The application we have implemented is based on a finite state machine to communicate the PC and the microprocessor. The methodology is quite straightforward, given the simplicity of our MCU which does not have neither an OS nor a file system.
The program runs over an infinite loop and listens to the PC that through UART until a command is received. Once at this point, the MCU processes the message and executes

a callback function, according to the content of the command that has been received. The microprocessor can be interpreted as a server, while a python script on the PC side performs the task of a client. The client application is capable of sending commands and documents to the MCU, as well as receiving results, like keys, signatures, performance metrics...
We designed a simple transmission protocol in order to communicate both ends and have a synchronous functioning of the whole system.

In a certain time instant, the application finds himself in a concrete state, where a handler function is executed. The next state is determined by the return value of this function. This methodology is followed indefinitely during the execution of the program.
The starting point is the IDLE state, which waits for the client to send a SPACE character (0x20) to launch the application. Once received, the application jumps to the D_IN state, where the the program waits for the commands corresponding to the algorithm that user demands to execute. The characters that refer to Key Generation, Signature and Verification are 0x00, 0x01 and 0x02 respectively. The D_RX state is responsible for checking the command received.

The KEY (`keygen_Handler()`), SIGN (`signature_Handler()`) or VERIF (`verification_Handler()`) states correspond to the three functions provided by the cryptosystem. After the program has returned from the handler of any of the mentioned states above, the application jumps back to the REST state, which allows the user to keep the application running or to terminate it. As the MCU does not have a file system, the keys and signatures

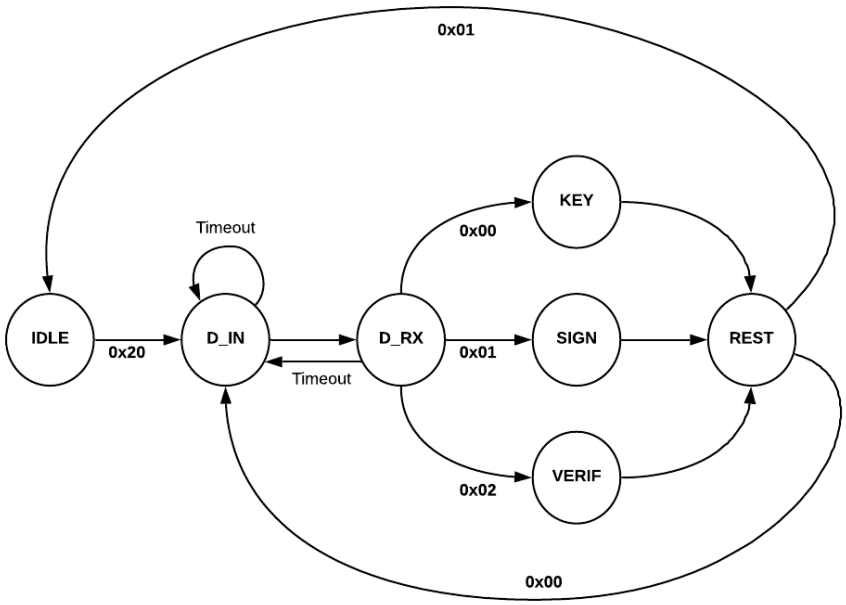

Figure 5.1: States diagram of the Rainbow embedded application

are stored in arrays of 8-bit integers that are declared as global variables in the code. Within the states SIGN and VERIF, the application waits for the client to send the entire message that has to be signed or verified before calling the corresponding handler, which implies that two more buffers are required to store this data. These arrays are allocated dynamically, once the whole messages have been received in the microprocessor side. Every time the application is terminated, all the buffers are freed from memory.

5.5 Performance Optimization

Regarding the performance of the Rainbow instances from Round 1 and Round 2 in our platform, we made the decision to optimize the execution time for the key generation, message signature and verification, in order to further increase the speed of the algorithm. The number of clock cycles elapsed to execute these functions is higher than in various post-quantum digital signature approaches. We want to prove that taking advantage of the scheme's construction features, the execution speed can be accelerated in order to get a modified Rainbow that processes the basic functions faster than other state-of-the-art digital signature alternatives in our platform.

It is important to notice that the implementation runs slower in our MCU than in other platforms with higher complexity, not only because of the faster clock but also because of the different instructions that are employed.
The microprocessor that is used is a reduced instruction set computing (RISC), resulting into smaller and simpler instructions fetched to the core. Therefore, in order to perform a specific task, the number of instructions needed is bigger in a RISC than in a CISC (complex instruction set computing) architecture, where longer and more complex instructions are executed. The instructions of the last ISA mentioned usually require various clock cycles to be executed. This is why CISC architectures are usually implemented in systems with a high frequency clock in the order of GHz. In our case, we have a lower frequency system clock combined with a huge amount of 1-cycle instructions, which ends up in a slower execution.

5.5.1 Data Collection

To quantify the performance of the optimized version of the algorithms that have been implemented, we analyze two different features: execution speed and stack memory usage.
Below is described the methodology followed for obtaining these metrics.

- **Stack Memory Usage**
  The stack usage is obtained by first filling the stack memory from a known address, given by the declaration of a variable, down to a concrete amount of bytes with known

dummy values. After that, we check the values stored from the lowest memory address storing a dummy value as starting point, to the first appearance of a different value. We claim the amount of stack used by subtracting the address corresponding to the first non-dummy value and the address of the declared variable before calling the function to evaluate. The stack usage is given in **Bytes** among all the tables that are exposed in this work. This sequential procedure is shown from left to right in 5.2.

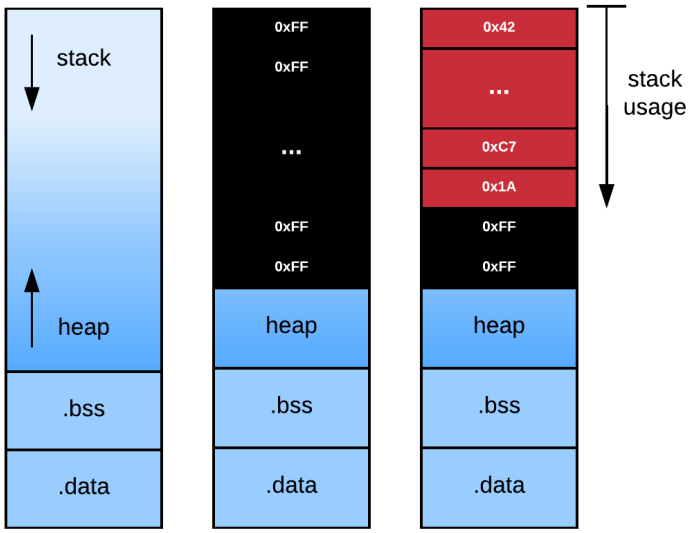

Figure 5.2: Stack usage measurement. The first picture shows the sectioned structure of the RAM. The second picture presents the memory with filled dummy values. The third picture illustrates the RAM layout before returning from a function.

- **Execution Time**
  For measuring the number of clock cycles spent in each function, we take advantage of the DWT_CYCCNT internal register of the Cortex-M4 [3]. This register increments by one the current stored value, every time a clock cycle elapses during the execution of a program. The number of cycles that can be represented range from 0 to $2^{32} - 1$. When the register overflows it is wrapped around 0.
  Therefore, in case that the function takes longer than 2^{32} clock cycles, we must know the number of times that the register overflows to obtain the total time. For this purpose, we use the internal SysTick timer. Using the HAL drivers functionalities, we can set a concrete number of elapsed ticks before the MCU throws an interruption calling `HAL_SYSTICK_Config`. In our case, we set this value to the maximum number that can be stored in the 24-bit SYST_CVR register[4].

[3] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439c/BABJFFGJ.html`
[4] `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0552a/Bhccbfia.html`

This register stores the current value of the SysTick Timer. The interruption handler is implemented to add one to `uwTick`, which is the global variable storing the number of times that the register has been overflowed.

Finally, we can compute the total number of clock cycles elapsed making use of both measurements with the following equation:

```
Clock_Cycles = numIRQ * maxTicks_Counter + current_Counter
```

5.5.2 Optimization Approach

The first step of the optimization process consists in evaluating the different factors that characterize the algorithms so that we can exploit them in order to achieve an optimized version. The fact that all the parameter sets evaluated are using the same hash function (SHA256), we can assume that no difference in performance between them will come from the hashing procedure, but from the size difference of the finite fields employed, as well as the parameters chosen to design the scheme.
As the latter factor is fixed to achieve a established level of security, we decide to focus on **finite field arithmetic** in order to increase the efficiency of the cryptographic functions provided by the Rainbow scheme.

The second step is based on exploring alternatives to the multiplication reference implementation, so that we can reduce the number of clock cycles to compute the operation. In order to do so, we took advantage of the following principles:

- Arithmetic in $GF(2^N)$ for large N can sometimes be too expensive in terms of computation, specially when using traditional multiplication methodologies. The use of the composite field representation can speed up the opeartions since the coefficients are represented by n-bit words, which is advantageous for the implementation of on microprocessors, specially when n is selected properly. However, this property can not be fully exploited as the size of the fields is relatively small.

- The use of look up tables allows us to substitute the reference implementation $log_2 n$-step Karatsuba-Ofman algorithm called every time a multiplication is performed, where n corresponds to the degree of the polynomials, for memory accesses to a precomputed table, containing the multiplications between every combination of elements from the underlying field. The construction time of the tables can be reduced if the degree of the irreducible polynomial in the tower structure equals 3 or 5 [SK10].

The look up table for multiplying $GF(2^N)$ elements is initialized once when the application is launched, so no overhead is generated. It is constructed using the Karatsuba-Ofman al-

gorithm to reduce the number of submultiplications to obtain the results.
For generating the $GF(16)$ table, we used logarithmic look-up tables to speed up the execution of this suboperations obtained after applying the Karatsuba decomposition method. The log-antilog tables compute the multiplications in the ground field. Therefore, we only used them to compute multiplications in $GF(4)$. The $GF(16)$ look up table is reused to tabulate the products from $GF(256)$.

The motivation to use log-antilog tables is given by the fact that a multiplication between two integers from a finite field $a \cdot b$ can be alternatively written as $log^{-1}(log(a) + log(b))$, which allows the user to compute the operation by simply performing three memory accesses to tables which size equals the number of elements of the field, and a modulo computation. Furthermore, the clock cycles can be reduced even more, by using an inverse logarithmic table twice the size of the original approach instead of the modulo. This methodology fits perfectly in small field multiplications, where doubling their size does not increase too much the number of elements in the antilog table.

It is required to ensure that after the modifications made to improve the efficiency, the scheme remains immune against timing attacks. As mentioned in 5.2.3, the tower field representation provides time-constant arithmetic opeartions between the components of the private key. Therefore it can be ensured that no side-channel information is leaked .
The use of look up tables can be exploited by performing cache-timing attacks, such as PRIME+PROBE and its variant FLUSH+RELOAD, which take advantage of the shared last-level cache [DHF$^+$10]. The time difference between the cache hits or misses to the look up tables leaks information about the structure of the secret key. This attacks consist in three stages. Firstly, the attacker flushes a cache line from the cache hierarchy. Secondly, he waits until the victim accesses that line. Then, the attacker reloads the content on that line and measures the access time. If the line has been stored in the cache (hit), the number of clock cycles to perform the task will be lower. Otherwise, it will take significantly longer, as the data will have to be brought from memory. This reveals the positions on the look up table that were accessed, and thus, the location where the elements are stored.

In our case, although using the look-up table approach, the security of our scheme will not get compromised, as the Cortex-M4 microprocessor does not have a cache memory structure.

Round 1

We compute the elements for the table of $GF(16)$ and $GF(256)$ using $GF(4)$ and $GF(16)$ look-up tables respectively, as it has been mentioned above.
Algorithms 4 and 5 describe the generation procedure of these tables in pseudo code. The functions `gf16_mul_lookupTable` and `gf4_mul_lookupTable` perform the memory accesses in order to get the result of the multiplications.

Therefore, we need to store a table of 256 (2^8) elements for Ia and 65536 (2^{16}) elements for Ic.

Algorithm 4 $GF(16)$Look-up Table Generation

Input: $GF(4)$ look-up table $gf4\_tab$
Output: $GF(16)$ look-up table $gf16\_tab$

1: $size \leftarrow 1 << 4$
2: **for** $i \leq size$ **do**
3: **for** $j \leq size$ **do**
4: $a_0b_0 \leftarrow$ `gf4_mul_lookupTable`$(x_0, y_0, gf4\_tab)$;
5: $a_1b_1 \leftarrow$ `gf4_mul_lookupTable`$(x_1, y_1, gf4\_tab)$;
6: $ab_x \leftarrow$ `gf4_mul_lookupTable`$(x_0 \bigoplus x_1, y_0 \bigoplus y_1, gf4\_tab) \bigoplus a_0b_0 \bigoplus a_1b_1$;
7: $a_1b_1\_2 \leftarrow$ `gf4_mul_lookupTable`$(a_1b_1, 2, gf4\_tab)$;
8: $gf16\_tab[16i + j] \leftarrow (ab_x \bigoplus a_1b_1) << 2) \bigoplus a_0b_0 + a_1b_1\_2$;
9: **return** $(gf16\_tab)$

Algorithm 5 Look-up $GF(256)$ Table Generation

Input: $GF(16)$ look-up table $gf16\_tab$
Output: $GF(256)$ look-up table $gf256\_tab$

1: $size \leftarrow 1 << 8$
2: **for** $i \leq size$ **do**
3: **for** $j \leq size$ **do**
4: $a_0b_0 \leftarrow$ `gf16_mul_lookupTable`$(x_0, y_0, gf16\_tab)$;
5: $a_1b_1 \leftarrow$ `gf16_mul_lookupTable`$(x_1, y_1, gf16\_tab)$;
6: $ab_x \leftarrow$ `gf16_mul_lookupTable`$(x_0 \bigoplus x_1, y_0 \bigoplus y_1, gf16\_tab) \bigoplus a_0b_0 \bigoplus a_1b_1$;
7: $a_1b_1\_8 \leftarrow$ `gf16_mul_lookupTable`$(a_1b_1, 8, gf16\_tab)$;
8: $gf256\_tab[256i + j] \leftarrow (ab_x + a_1b_1) << 4) \bigoplus a_0b_0 \bigoplus a_1b_1\_8$;
9: **return** $(gf256\_tab)$

There is an alternative for $GF(256)$ multiplications to save space in the SRAM, given that the size of the table used of size 2^{16} in addition to the public and secret key buffers, constrain drastically the amount of memory available left for the execution of the cryptographic procedures of key generation, signing and verification. Given that the Karatsuba-Ofman decomposition of the $GF(256)$ multiplication consists in 4 $GF(16)$ multiplications, we can perform 4 memory accesses to the $GF(16)$ table instead of 1 access to the $GF(256)$.

Choosing between the two variants depends on the user's interest. The $GF(256)$ table option is faster but needs a huge table to store all the values. Even though accessing a bigger table is slower than checking entries in a smaller table, it is still faster than the second option, where 4 accesses are required. In this case, the table's size used is very small compared to the other alternative, thus saving a great amount of memory space. In this case, the difference in size between the tables of both methods is 65280 bytes, given by $2^{16} - 2^8$.

Round 2

In this case, the reference implementation of Ia_Classic computes multiplications on a single 32-bit variable composed by 8 $GF(16)$ elements to make an efficient use of the variables. We want to apply the look up table approach employed on the Round 1 parameter sets in order to speed up this procedure. In order to do so, we build a look up table of $GF(16)$ that consists of precomputed multiplications between all the possible numbers that can be formed by a combinations of two $GF(16)$ integers (0 to 255) and a single $GF(16)$ element. Thus, we obtain a table of 4096 bytes, given by 256×16. The generation procedure is described in algorithm 6. To compute the multiplication, instead of splitting 8 times the 32 bit variable into elements of $GF(16)$ and then apply the recursive Karatsuba-Ofman, we save 4 splits in the generation of the look up table. Therefore, our optimized method to compute 8 multiplications in $GF(16)$ consists in 4 splits and 4 memory accesses.

Algorithm 6 Look-up $GF(256)$ Table Generation (32-bit version)

Input: $GF(16)$ look-up table $gf16\_tab$
Output: $GF(16)$ look-up table $gf16\_tab\_v32$

1: $size1 \leftarrow 1 << 4$
2: $size2 \leftarrow 1 << 8$
3: **for** $i \leq size1$ **do**
4: **for** $j \leq size2$ **do**
5: $j_0 \leftarrow j \ \& \ 7$;
6: $j_1 \leftarrow (j \ \& \ 240) >> 4$;
7: $mul_0 \leftarrow$ `gf16_mul_lookupTable`$(j_0, i, gf16\_tab))$;
8: $mul_1 \leftarrow$ `gf16_mul_lookupTable`$(j_1, i, gf16\_tab))$;
9: $gf16\_tab\_v32[16j + i] \leftarrow mul_0 \bigoplus (mul_1 << 4)$;
10: **return** $(gf16\_tab\_v32)$

5.5.3 Reference Implementation Performance

To analyze the variation in the efficiency of the optimized version of Rainbow that we implemented, we first need to evaluate the performance of the reference implementation.
For quantifying the performance we based ourselves in the one hand, on the size of the keys, signatures and hash values generated, to be aware of how much memory is employed for their storage which is exposed in (Table 5.3). The length of these elements is directly determined by the parameters set that is chosen to construct the scheme.

On the other hand, we extract the number of clock cycles elapsed and the amount of Bytes in the stack employed by the key generation, signing and verification functions for the different parameter sets evaluated. The results are presented in Tables 5.4, 5.5 and 5.6, corresponding to Ia, Ic and Ia_Classic respectively.

Set	Parameters	Public Key	Private Key	Hash Size	Signature
Ia	$(GF(16), 32, 32, 32)$	152097	100209	256	512
Ic	$(GF(256), 40, 24, 24)$	192241	143385	256	832
Ia_Classic	$(GF(16), 32, 32, 32)$	148992	92960	256	512
Ia_Cyclic	$(GF(16), 32, 32, 32)$	58144	92960	256	512
Ia_CompCyclic	$(GF(16), 32, 32, 32)$	58144	64	256	512

Table 5.3: Key, Hash and Signature Sizes in Bytes for Ia and Ic parameter sets from Round 1 and the different variants from Ia in Round 2.

Ia				
Optimization Level	**Metrics**	**Key Gen**	**Sign Gen**	**Sign Verif**
Og	Cycles	34095356078	52853987	36995702
	Memory	1618752	24576	6912
O1	Cycles	28182442856	16352581	14968190
	Memory	1619520	24736	7040
O2	Cycles	28182105383	16454817	16196426
	Memory	1619584	24576	6368
Ofast	Cycles	25023292079	16216028	14800683
	Memory	1620800	24032	6304

Table 5.4: Performance Metrics of the reference Rainbow procedures in the **Ia** parameter set

Ic				
Optimization Level	**Metrics**	**Key Gen**	**Sign Gen**	**Sign Verif**
Og	Cycles	67861477282	60182653	60384694
	Memory	1162944	23168	6720
O1	Cycles	57182539666	52380394	52522017
	Memory	1164288	23808	7296
O2	Cycles	58163239932	54790748	54953725
	Memory	1165504	23072	6368
Ofast	Cycles	58250523705	55102359	55704486
	Memory	1167616	21632	6368

Table 5.5: Performance Metrics of the reference Rainbow procedures in the **Ic** parameter set

Ia_Classic				
Optimization Level	**Metrics**	**Key Gen**	**Sign Gen**	**Sign Verif**
Og	Cycles	151223541	2157586	1981264
	Memory	1792	7744	6080
O1	Cycles	144266399	1960422	1715855
	Memory	2304	7904	6240
O2	Cycles	135691905	1815468	1638395
	Memory	2048	8416	6304
Ofast	Cycles	134354438	-	1618523
	Memory	2752	-	6560

Table 5.6: Performance Metrics of the reference Rainbow procedures in the **Ia_Classic** parameter set

5.5.4 Optimized Implementation Performance

In order to evaluate the impact of the changes applied, we analyzed the time spent by both the reference and optimized multiplication algorithms isolated, in each of the different parameter sets employed. After that, we measured the number of clock cycles elapsed during the three different cryptographic procedures to test the influence of the new multiplication approach. The stack memory usage is also presented in the tables.
We obtained the time in milliseconds computing the division between the number of clock cycles elapsed and the system's clock frequency, which in our case is 120 MHz.

We are using most of the available compiler optimization flags for ARM microprocessors, in order to explore as much alternatives as possible to minimize the size of the code while squeezing the execution time. It is not possible to exactly determine the metrics that would present the different cryptographic functions by looking at the isolated multiplications table, as the compilation process takes into account all the code in order to optimize its execution. Therefore, there is no strict relationship between the two measurements, given that there exists a huge different between both cases in the amount of code visible by the compiler.

Tables 5.7 references the metrics extracted from the optimized Rainbow built using the Ia set from Round 1. The same is exposed in Table 5.8 for the Ic set. Regarding Round 2, the performance results using the Ia_Classic parameters set are shown in tables 5.9.
It can be observed that signature procedure on the Ia_Classic parameter set with the Ofast compiler flag is not shown. This is because the implementation was falling into undefined behaviour when selecting this optimization option.

Ia				
Optimization Level	**Metrics**	**Key Gen**	**Sign Gen**	**Sign Verif**
Og	Cycles	2248623368	5362571	4801500
	Memory	1617984	24192	6464
O1	Cycles	3525035725	4323043	3701322
	Memory	1618688	24320	6656
O2	Cycles	1295429175	3611927	2936378
	Memory	1618816	24224	5984
Ofast	Cycles	1446131173	3428842	3185447
	Memory	1621120	25440	5984

Table 5.7: Performance Metrics of the optimized Rainbow procedures in the **Ia** parameter set

Ic				
Optimization Level	**Metrics**	**Key Gen**	**Sign Gen**	**Sign Verif**
Og	Cycles	6939685249	4093260	4211083
	Memory	1162048	22784	6272
O1	Cycles	5295748895	3287900	3021377
	Memory	1162816	22848	6464
O2	Cycles	4218783615	3122501	2889175
	Memory	1162880	22240	5600
Ofast	Cycles	4185819248	3186323	3052783
	Memory	1163328	22624	5600

Table 5.8: Performance Metrics of the optimized Rainbow procedures in the **Ic** parameter set

Ia_Classic				
Optimization Level	**Metrics**	**Key Gen**	**Sign Gen**	**Sign Verif**
Og	Cycles	178936017	2472074	2292619
	Memory	1856	7872	6208
O1	Cycles	164643555	2362744	2065065
	Memory	2304	8256	6592
O2	Cycles	146019481	2930942	1321850
	Memory	2176	8640	6528
Ofast	Cycles	115099104	-	1376828
	Memory	2688	-	6528

Table 5.9: Performance Metrics of the optimized Rainbow procedures in the **Ia_Classic** parameter set

6 Results Analysis

6.1 Key Generation

In Figure 6.1, it is shown the number of clock cycles that are elapsed during the Key Generation process on the three parameters sets that we are evaluating, for different compiler optimization flags. Looking at the bar graph, we can claim that the interpolation method employed in Round 1, is much slower than the matrix multiplication approach from Round 2. In the modified versions of Rainbow of Ia and Ic, the multiplications are made as a single memory access, which results into a notable increase in the speed of the function, that originally uses the Karatsuba-Ofman algorithm until $GF(2)$ for computing the operation.

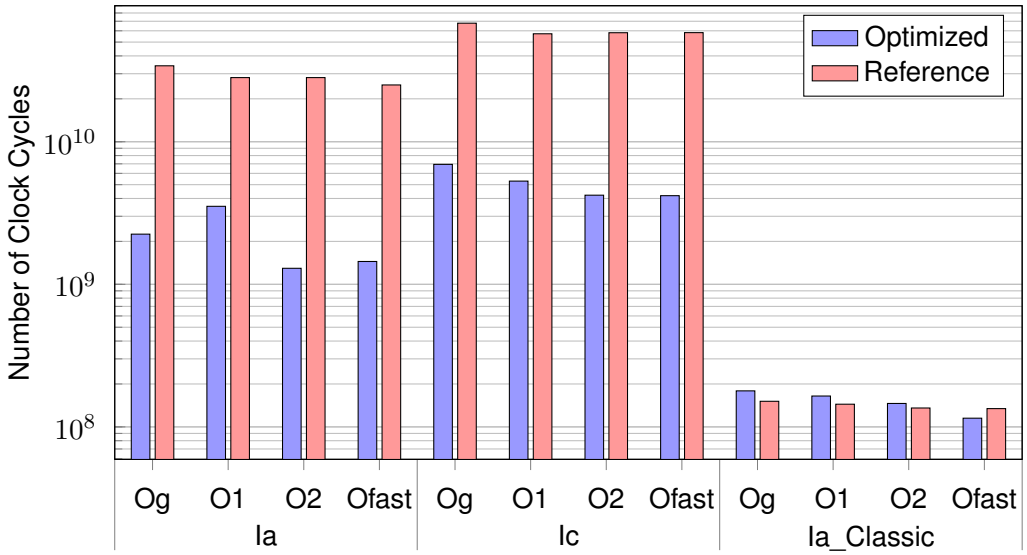

Figure 6.1: Optimized and Reference Rainbow Performance for Key Generation on Ia, Ic and Ia_Classic.

In the Ia_Classic set, the multiplications in the reference implementation are made on 32-bit variables, which are decomposed into $GF(4)$ before being bit-wise multiplied, which turns out to be very efficient because of the word length of our hardware architecture. In this case, the optimization approach that we proposed consists in 4 memory accesses to a table

of 4096 bytes, which also requires the decomposition of the entire variable into 4 elements of $GF(256)$ and the reassembly of the variable to return. This procedure's overhead results into a longer generation time of the keys, compared to the reference implementation for some of the compiler optimization flags used.

6.2 Signature Generation

Figure 6.2 shows the bar graphs for signature generation. The overall behaviour of the different parameter sets is very similar to the key generation process. The difference in the number of clock cycles is increased 9 times for Ia and 11 for Ic. In the Ia_Classic, the optimized approach is not able to reduce the time respect the reference implementation. Between the optimized versions of both Round 1 and Round 2, the speed difference is about 2×10^6 clock cycles.

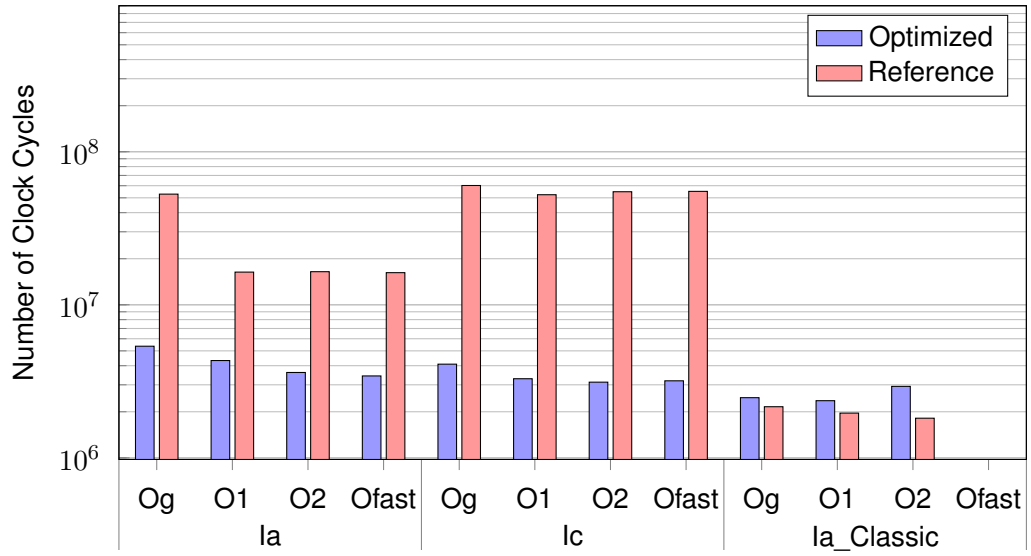

Figure 6.2: Optimized and Reference Rainbow Performance for Signature Generation on Ia, Ic and Ia_Classic.

6.3 Signature Verification

The gain of verification time between the reference and optimized versions of Ia_Classic is minimal as can be observed in Figure 6.3, equally to the key and signature generation, given that the multiplication by table look ups has no significant run-time difference at all,

in comparison to the 32-bit multiplication approach implemented in the reference version. We are able to improve the speed of this set only with -O2 and -Ofast optimization flags. Otherwise, for the sets from Round 1, the execution time is reduced by the same factor as the achieved in the signature procedure.

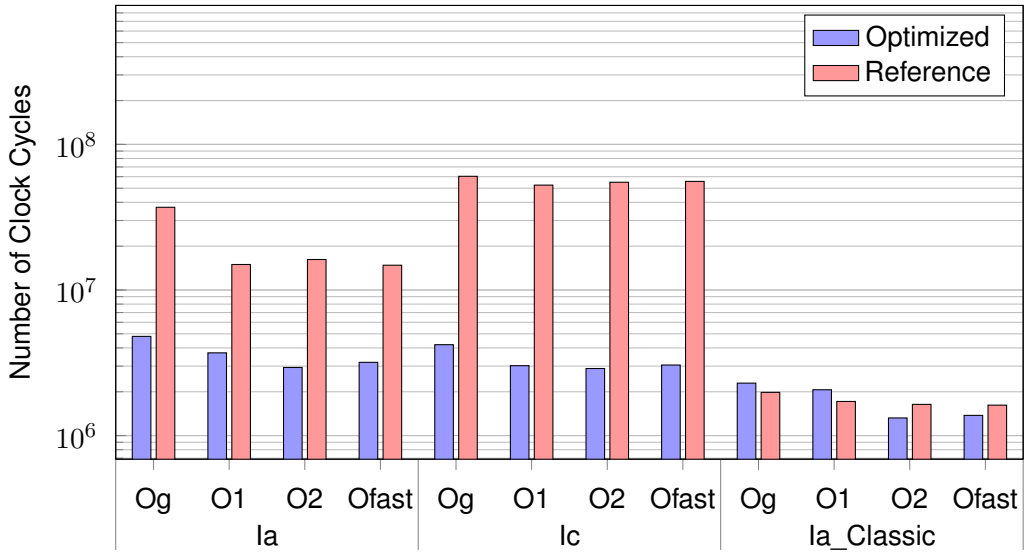

Figure 6.3: Optimized and Reference Rainbow Performance for Signature Verification on Ia, Ic and Ia_Classic.

6.4 Comparison with other Cryptosystems

To evaluate the functioning of the Rainbow cryptosystem in a global scene, we provide a quantitative comparison of the performance achieved during the signature and verification functions, with timing benchmarks for various state-of-the-art post-quantum signature schemes [KRSS]. These algorithms are the Lattice-based DilithiumIII[DKL$^+$17] and qTesla-II [BAA$^+$19], the Hash-based SPHINCS+sha256-128f[BDE$^+$19], and the multivariate LUOV-II [BPSV19]. In order to give a fair comparison, we have selected concrete versions of the post-quantum techniques whose security level [BBB$^+$12] is equal to the achieved by the Rainbow parameter sets that are evaluated in this thesis: 128-bit post quantum security (levels 1 and 2 of the NIST requirements). The Rainbow metrics shown in the plots correspond to the best result achieved, in terms of speed, for each of the parameter sets.

Although, many aspects should be taken into account to offer an exact comparison, such as computational assumptions, implementation details or constant-time against non-constant-time approaches, we intend to give a precise differentiation between different algorithms that compete to become the quantum secure digital signature standard.

Methodology. The timings for Rainbow Ia, Ic and Ia_Classic sets are obtained with the core running at 16MHz, while the measurements for the other algorithms are acquired at 24MHz. However, the benchmarks have been generated in the same exact platform (Cortex-M4). Even though the times are obtained using different clock frequencies, the fact that the instruction set architecture of the MCU's is the same in all cases (ARMv7-M), allows the scaling of the time to give a highly precise approximation of the milliseconds elapsed in a common clock frequency basis. The measurements are taken at a certain frequency where the wait cycles due to the speed of the memory controller are not added to the elapsed time.

Algorithm	**Sign Time** (ms)	**Verif Time** (ms)	**Pk** (KB)	**Sk** (KB)	**Signature** (B)
Ia	0.028	0.026	152.097	100.209	64
Ic	0.026	0.024	192.241	143.385	104
Ia_Classic	0.015	0.013	148.992	92.960	64
qTesla-II	0.156	0.027	2.336	1.600	2144
LUOV-II	1.133	0.898	12.100	0.032	311
Dilithium-III	0.077	0.019	1.472	-	2701
SPHINCS+ 128	4.349	0.174	0.032	0.064	16976

Table 6.1: List of the algorithms evaluated their corresponding performance metrics in terms of execution time and size.

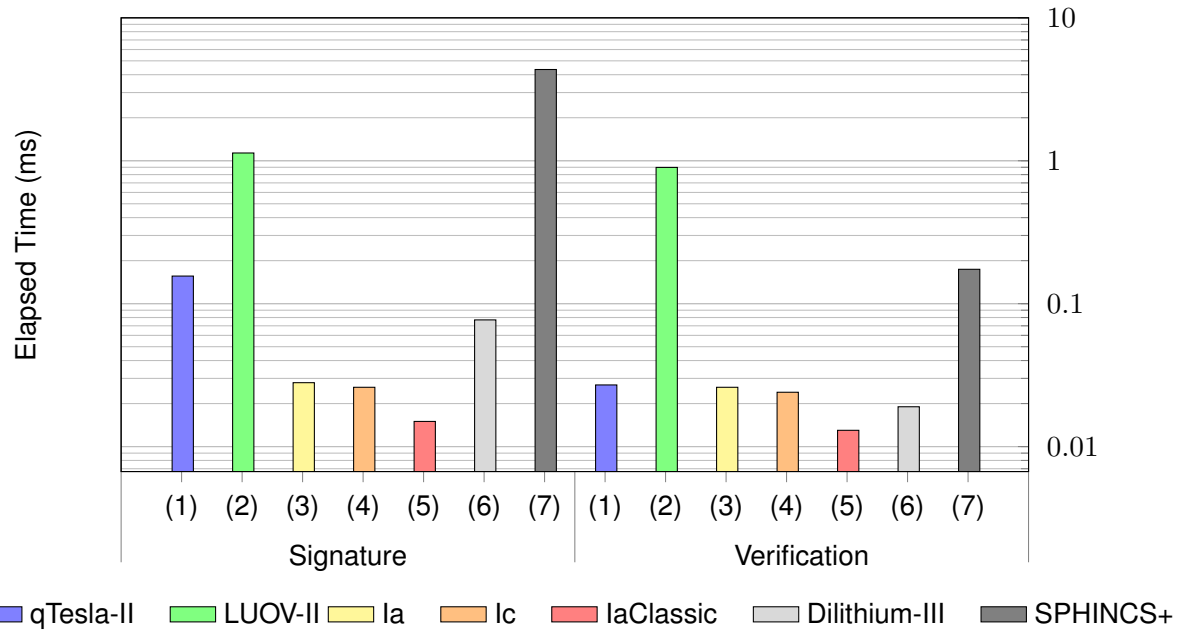

Figure 6.4: Execution time comparison between different public key algorithms during signature and message verification.

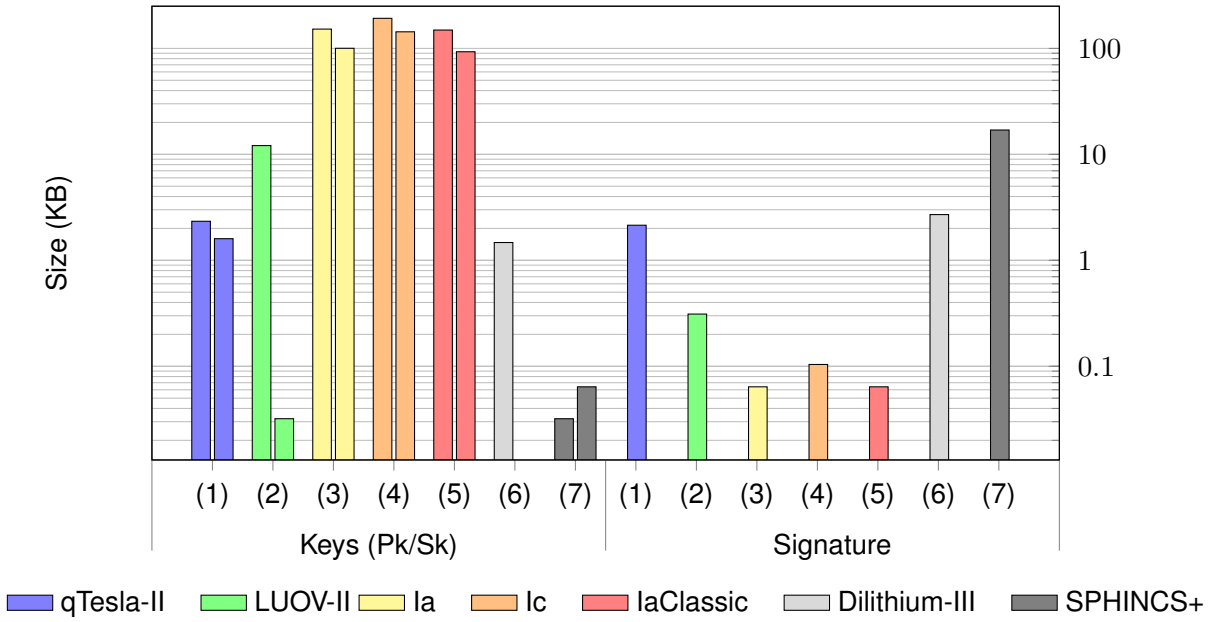

Figure 6.5: Size of the public key and secret key on the left and signature on the right. The measurements are expressed in Kbytes.

Analyzing the bar plot in Figure 6.4, we can claim that during the signature procedure, all the different versions of Rainbow are faster than any of the other post-quantum approaches presented, concretely by 0.062 ms in case of Ia_Classic and 0.049 ms and 0.051 ms in case of Ia and Ic respectively, compared to the signing speed of Dilithium-III, which is the fastest algorithm right after the Rainbow schemes.
The Ia_Classic Rainbow is still the fastest in verification speed, although in this case, Dilithium-III overcomes by 0.015 ms the sets Ia and Ic. The lattice-based qTesla is quite fast during verification speed unlike in the signature case. It is interesting also to compare the different behaviour between Rainbow and the multivariate approach LUOV-II, which is a single layer version of the former, that employs a different representation of its keys to drastically reduce the amount of memory needed for their storage. This has a huge impact over the speed of both the signature and verification cases.

Regarding the size of the keys and signatures generated by the cryptographic techniques in Figure 6.5, we can affirm that Rainbow generates the smallest signatures, although the size of its keys its about hundreds of kilobytes, which are very big compared to the ones provided by the other algorithms. The SPHINCS+sha256-128f cryptosystem needs the smallest amount of space to store the pair of public and private keys (96 Bytes). The biggest signatures are generated by the the lattice-based Dilithium-III (2.7 KBytes) and the hash-based SPHINCS+sha256-128f (16.9 KBytes).

Based on the observations, we can conclude that there clearly exists a trade-off between the execution time of the different procedures and the memory needed to store the key pair and the signature. The bar plots show that there is not any approach from the ones analyzed, that actually has a good balance between the aspects evaluated.
We can hold that the Rainbow cryptosystem offers the fastest signature and message verification among the approaches compared. It also has the shortest signatures even though the size of its keys is very large in contrast to the other authentication cryptographic solutions. This results are based on the fact that all cryptosystems share the same level of security. It would be interesting to evaluate how the speed and the memory amount required vary depending on the security level of the scheme as the choice of the parameters is not done equal in all the algorithms but according to its own structure.

There are still a few other signature schemes submitted to the NIST competition that have not been included in this comparison: Falcon [PFH$^+$19], GeMSS [CFMR$^+$19], MQDSS [SCH$^+$19] and Picnic [ZCD$^+$19]. The reason why we dismissed this approaches is because they are unable to fit in the memory of our target device, due to the large keys generated and the extensive use of the stack that they perform.

7 Conclusions

In this thesis, we evaluated the efficiency of the Rainbow multivariate scheme in an ARM Cortex-M4 microprocessor, in order to study the feasibility of its implementation on a constrained device.
It has been shown that the level of security that Rainbow can provide is notably constrained due to the memory limitations of our platform, even though the parameter sets according to the two first security categories proposed by NIST, are still suitable options. Rainbow has been proven to be the fastest in both signature generation and verification, compared to a group of several state-of-the-art post-quantum digital signature techniques, sharing the same security level (category 1). We also manifested the limitation that implies the size of the keys generated by this $\mathcal{MQ}$-based algorithm, which is the main obstacle faced during the implementation in limited-resource platforms.
We proposed an optimized version of the Rainbow reference constructions that were able to fit in our MCU, by implementing a look-up table-based approach, exploiting the characteristic architecture of our platform and the construction principles of the scheme. The modifications resulted in the reduction of the signing and verifying times by a factor of 17.5 and 19 respectively, in two out of three cases. In consequence of the optimization approach followed, we implemented a more efficient Rainbow instance, providing category 2 security, with a faster signature generation than all the different post-quantum cryptographic solutions analyzed.

7.1 Further Improvements

Rainbow provides notably short signatures due to its construction principle, even though the public and private keys are too big to fit all the possible parameter sets available in a constrained device. Once known that this approach provides short signatures which are generated at a relatively high speed compared to other post-quantum families, the next step would consist in reducing the amount memory required to store the keys. Below are some options that have already been proposed for this purpose.

- **Private Key :** The affine transformations and the central map from the secret key are composed by random values from the same finite field generated by a PRNG. Instead of storing all the bytes corresponding to the coefficients of the polynomials, we can simply store a n-bit seed, which is enough to always generate the same random num-

bers. The drawback of this technique is the efficiency reduction during the signature generation, as all the elements that constitute the private key have to be generated every time that the key has to be employed.

- **Public Key :** The size of the Rainbow construction is tightly related to the parameter set that is chosen to build the scheme. Exhaustive research has been made on the optimized selection of parameters, in order to find combinations between them that are able to reduce the size of the keys without downgrading the level of security.
  Another option to efficiently store the key is based on Macauley matrices. The technique consists in inserting a structured submatrix into the public map coefficient Macauley matrix. This construction provides the possibility to the user to fix most part of the public key so the central map of the scheme is computed out of it. Partially circulant matrices or linear recurring sequences are two alternatives for choosing the composition of the structured matrix. Furthermore, depending on the submatrix structure, the verification process of the scheme can be speed up.

As mentioned several times along the thesis and further proven in Section 6.4, there exists trade-off that has to be respected between the memory needed for storing the keys and the speed at which the key generation, message signature and verification are executed. Employing the techniques proposed in different research papers for reducing the memory footprint, it is shown that the execution time elapsed for signing and verifying messages gets always longer.
Given the advantage that has Rainbow in terms of speed, in front of other post-quantum approaches, this algorithm has still margin to improve the memory efficiency while still being one of the fastest cryptographic solutions among quantum secure digital signature proposals.

Bibliography

[BAA+19] Nina Bindel, Sedat Akleylek, Erdem Alkim, Paulo S. L. M. Barreto, Johannes Buchmann, Edward Eaton, Gus Gutoski, Juliane Kramer, Patrick Longa, Harun Polat, Jefferson E. Ricardini, and Gustavo Zanon. Lattice-based digital signature scheme qtesla, 2019. https://github.com/qtesla/qTesla.

[BBB+12] E B Barker, W C Barker, W E Burr, W T Polk, and M E Smid. Recommendation for key management, part 1 :. Technical report, 2012.

[BDE+19] Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Andreas Hülsing, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Christian Rechberger Ruben Niederhagen, Joost Rijneveld, and Peter Schwabe. Sphincs+ – submission to the nist post-quantum project, 2019. https://sphincs.org/resources.html.

[BDKR03] Raghav Bhaskar, Pradeep Dubey, Vijay Kumar, and Atri Rudra. Efficient galois field arithmetic on simd architectures. pages 256–257, 01 2003.

[BPSV19] Ward Beullens, Bart Preneel, Alan Szepieniec, and Frederik Vercauteren. Submission of the luov signature scheme to the nist pqc project, 2019. https://github.com/WardBeullens/LUOV.

[CDF02] Nicolas T. Courtois, Magnus Daum, and Patrick Felke. On the security of hfe, hfev- and quartz. Cryptology ePrint Archive, Report 2002/138, 2002. https://eprint.iacr.org/2002/138.

[CFMR+19] A. Casanova, J.-C. Faugère, G. Macario-Rat, J. Patarin, L. Perret, and J. Ryckeghem. Gemss. technical report, 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[Che06] Y Chen. An implementation of pmi+ on low-cost smartcard. *Master's thesis, National Taiwan University*, 2006.

[DBM12] Philippe Dreesen, Kim Batselier, and Bart De Moor. Structured matrices and solving multivariate polynomial equations, 2012.

[DHF+10] Jean-Luc Danger, Youssef El Housni, Adrien Facon, Cheikh T. Gueye, Sylvain Guilley, and Sylvie Herbel. On the performance and security of multiplication in gf(2^N). 2010.

[DKL+17] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-dilithium – submission to the nist post-quantum project, 2017. `https://pq-crystals.org/dilithium/index.shtml`.

[DPSY19] Jintai Ding, Ming-Shing Chen Albrecht Petzoldt, Dieter Schmidt, and Bo-Yin Yang. Rainbow. technical report, 2019. `https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions`.

[DS05] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In John Ioannidis, Angelos Keromytis, and Moti Yung, editors, *Applied Cryptography and Network Security*, pages 164–175, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[DYC+08] Jintai Ding, Bo-Yin Yang, Owen Chen, Ming-Shing Chen, and Doug Cheng. New differential-algebraic attacks and reparametrization of rainbow. Cryptology ePrint Archive, Report 2008/108, 2008. `https://eprint.iacr.org/2008/108`.

[Eyu15] Can Eyupoglu. Performance analysis of karatsuba multiplication algorithm for different bit lengths. *Procedia - Social and Behavioral Sciences*, 195:1860–1864, July 2015.

[GCL92] Keith O Geddes, Stephen R Czapor, and George Labahn. *Algorithms for computer algebra*. Springer Science & Business Media, 1992.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[Gro96] Lov K Grover. A fast quantum mechanical algorithm for database search. *arXiv preprint quant-ph/9605043*, 1996.

[KPG99] Aviad Kipnis, Jacques Patarin, and Louis Goubin. Unbalanced oil and vinegar signature schemes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 206–222, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[KRSS] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stof-

felen. PQM4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[Laz83] D. Lazard. Gröbner bases, gaussian elimination and resolution of systems of algebraic equations. In J. A. van Hulzen, editor, *Computer Algebra*, pages 146–156, Berlin, Heidelberg, 1983. Springer Berlin Heidelberg.

[LSY+16] L.Chen, S.Jordan, Y.K.Liu, D.Moody, R.Peralta, R.Perlner, and D.Smith-Tone. Report on post-quantum cryptography. 2016.

[Pat96] Jacques Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): Two new families of asymmetric algorithms. In Ueli Maurer, editor, *Advances in Cryptology — EUROCRYPT '96*, pages 33–48, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[PBB10] Albrecht Petzoldt, Stanislav Bulygin, and Johannes Buchmann. Cyclicrainbow – a multivariate signature scheme with a partially cyclic public key. In Guang Gong and Kishan Chand Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, pages 33–48, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[PBD14] Jaiberth Porras, John Baena, and Jintai Ding. Zhfe, a new multivariate public key encryption scheme. In Michele Mosca, editor, *Post-Quantum Cryptography*, pages 229–245, Cham, 2014. Springer International Publishing.

[PCG01] Jacques Patarin, Nicolas Courtois, and Louis Goubin. Flash, a fast multivariate signature algorithm. In David Naccache, editor, *Topics in Cryptology — CT-RSA 2001*, pages 298–307, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[PFH+19] Thomas Prest, Pierre-Alain Fouque, Jeffrey Hoffstein, Paul Kirchner, Vadim Lyubashevsky, ThomasPornin, Thomas Ricosset, Gregor Seiler, William Whyte, and Zhenfei Zhang. Falcon. technical report, 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[rou17] Post-quantum cryptography: Round 1 submissions, 2017. https://csrc.nist.gov/projects/post-quantum-cryptography/round-1-submissions.

[rou19] Post-quantum cryptography: Round 2 submissions, 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[SCH+19] Simona Samardjiska, Ming-Shing Chen, Andreas Hulsing, Joost Rijneveld,

and Peter Schwa. Mqdss. technical report, 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.

[Sho99] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Review*, 41(2):303–332, January 1999.

[SK10] Erkay Savas and Cetin Koc. Finite field arithmetic for cryptography. *IEEE Circuits and Systems Magazine*, 10(2):40–56, 2010.

[TC01] T.Moh and Jiun-Ming Chen. On the goubin-courtois attack on ttm. Cryptology ePrint Archive, Report 2001/072, 2001. https://eprint.iacr.org/2001/072.

[WYHL06] Lih-Chung Wang, Bo-Yin Yang, Yuh-Hua Hu, and Feipei Lai. A "medium-field" multivariate public-key encryption scheme. In David Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, pages 132–149, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[YC05] Bo-Yin Yang and Jiun-Ming Chen. Building secure tame-like multivariate public-key cryptosystems: The new tts. In Colin Boyd and Juan Manuel González Nieto, editors, *Information Security and Privacy*, pages 518–531, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[YDH+13] Takanori Yasuda, Xavier Dahan, Yun-Ju Huang, Tsuyoshi Takagi, and Kouichi Sakurai. Mq challenge: Hardness evaluation of solving multivariate quadratic problems. https://eprint.iacr.org/2015/275.pdf, 2013.

[ZCD+19] Greg Zaverucha, Melissa Chase, David Derler, Steven Goldfeder, Claudio Orlandi, Sebastian Ramacher, Christian Rechberger, Daniel Slamanig, Jonathan Katz, Xiao Wang, and Vladmir Kolesnik. Picnic. technical report, 2019. https://csrc.nist.gov/projects/post-quantum-cryptography/round-2-submissions.